eStream Cache Manager Straw Man Proposal

Version 0.2

[redacted]

Purpose

The purpose of this document is to serve as the basis for the design of the eStream Cache Manager. As a straw man, this document is meant to serve as the basis for discussion, and anything here is subject to change. Assuming there are no major concerns with this document, I will proceed with producing a low level design for the cache manager.

Requirements in Brief

Support > 2GB client cache, possibly across multiple drives
Provide some level of protection against piracy, via both the file system and the cache
Fast lookup for what is in the cache and where to find it
Support automatic and user-specified cache size policies

As far as cache size goes, I think that it is reasonable for eStream 1.0 for the cache to be limited to one disk partition and 2GB of space, but the design should allow for very large caches (spanning more than one file and possibly more than one drive letter.) Note that if the cache is greater than 2GB in size, it cannot be mapped into the address space of a single process under NT/2000 on x86.

Cache Organization

The cache will be contained in 2 or more files. One file will contain the cache indices, and one or more files will contain the data blocks for cached files. (More than one cache data file may be required if the cache is larger than the largest file allowed on the native file system.) This allows us to keep the cache index file memory mapped and only map the data file(s) if there is enough memory space to do so.

Data Blocks

The cache data file will contain data pages frome the file system 4k in size.

Data will be stored in the cache uncompressed to allow easy page retrieval.

Cache Index

The cache index will be a b-tree. The key for the lookup will be the file id and page number requested. Keys in the b-tree are the set { volume #, file #, starting page, # of pages }. A lookup will succeed when the volume number and file number match, and the requested page is in the range from starting page to starting page + # of pages. The data stored for that key will be the offset into the cache for the beginning of the run. As is described in the file system proposal, the file number and starting pages are each 32 bits long. I propose making the starting page a 48 bit number and the number of pages a 16 bit number. This allows us to have a very large total cache and reasonable sized runs of contiguous pages in the cache.

Free space in the cache will have to be managed. Free blocks can be placed into a specially identified "free space file" in the index. Some auxiliary data structures may be convenient to make searching for a region of free space of a particular size.

Metadata for a file would be stored in the cache. It would be indexed by page number -1 in the index.

Cache Replacement Policy

For simplicity, I propose that the cache manager evict entire files from the cache when it decides that it needs to clear room in the cache. (Of course, any fragmentary file that is in the cache can be evicted.) We should implement LRU for cache replacement, so we will evict files for apps that have not been run recently.

One Cache Per System

Administrator priviledges are required to install eStream. While various users on a system might have conflicting desires about eStream configuration, such as the size of the cache, I think that it is reasonable to have a policy where the adminstrator controls the setup of the eStream client. By limiting the cache to one per system, we eliminate any ambiguity about cache use in a multiuser environment.

Profiling and Prefetching

Profiling and prefetching have been broken out as a separate component in the client. It will be described elsewhere. It is expected that while the profiler/prefetcher will want access to the cache data structures (i.e. it wants to know what's already in the cache), the logic associated with prefetching is not logically tied to the cache manager, and should thus be separated.

Future Directions

Compression of the cache could potentially be a big win. We could provide cache compression similar to the way that NTFS provides file compression - we compress some number of blocks at a time (e.g. 16) and only store the compressed data when it saves at least one block of storage. Caching of data on disk can sometimes be a performance win, since decompressing the data can be faster than trasferring it on disk if the disk is slow enough.

eStream File System Driver Low Level Design

version 1.4
Curt Wohlgemuth
[redacted]

Functionality

The eStream Windows NT/2000 File System Driver (EFSD) is a kernel-mode file system driver to which file requests will be forwarded by the NT I/O Manager. It is the point of contact for users to access files on an eStream server. It works with the NT File Cache Manager to insure that kernel file caching is available for eStreamed files.

The Windows 98 EFSD is almost certainly to be very different from the driver for WNT and Win2K, and will not be described here.

In this document, I'm assuming that the EFSD communicates closely with the eStream cache manager (ECM) to perform the various file system requests. There may in fact be several components—if for example the ECM is broken into sub-components. Also, this document assumes that the ECM is in user mode; if this ends up in kernel mode, we will need significant changes to the interfaces to it.

Data type definitions

File handle

A file handle passed between the EFSD and the ECM is defined by the ECM:

```
typedef uint32 EFSDFileHandle;
```

Names

All file and directory names will be passed as counted Unicode strings, basically as defined by the NT header files. Note, however, that in NT the Buffer field is a pointer; for our purposes in communicating with the ECM, it's a NULL-terminated variable length array:

```
typedef struct _UNICODE_STRING {
   USHORT Length;
   USHORT MaximumLength;
   USHORT Buffer[1]; // NULL-terminated, 2-byte
                     // characters
} UNICODE_STRING;
```

Time stamps

The NT standard time format is a signed 8-byte integer representing the number of 100-nanosecond intervals since January 1, 1601. These time stamps will be tracked for files and directories:

- Creation time
- Modification time

File attributes

File attributes are contained in an unsigned 4-byte integer. This subset of attributes from Windows NT will be supported:

```
FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
```

These attributes are not supported:

```
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DEVICE
FILE_ATTRIBUTE_SPARSE_FILE
FILE_ATTRIBUTE_REPARSE_POINT
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_OFFLINE
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
FILE_ATTRIBUTE_ENCRYPTED
```

File size

File size will be represented as a 4-byte unsigned integer. Since sparse files are not supported, there will only be one file size passed between the ECM and the EFSD.

Metadata

This structure is defined to pass file and directory metadata between the EFSD and the ECM:

```
typedef struct { // 24 bytes, 4-byte aligned
   int64 CreateTime;
   int64 ModifyTime;
   uint32 FileSize;
   uint32 Attributes;
```

```
} Metadata;
```

Interface definitions

The EFSD is called by several different components, including

- the NT Executive (I/O Manager, Virtual Memory Manager), for standard file system requests
- the ECM, for these same file system requests, and to invalidate cached pages for coherency
- the client start software, to start and stop the EFSD

The EFSD supports standard FSD interfaces to the NT Executive modules; not all possible interfaces are supported, because the eStream file system is relatively low-functionality (compared to NTFS, for example).

The following file system requests will be supported; the interfaces for them will not be shown here, as they can be found in the DDK documentation.

- Create IRPs, for both new and existing files
- Cleanup and Close IRPs
- Read and Write IRPs:
    - synchronous and asynchronous
    - cached and non-cached
    - paging and non-paging
- Fast I/O reads and writes (with buffers or MDLs)
- File information (get and set) IRPs
- Directory query IRPs
- Volume information (get and set) IRPs
- File system information (get and set) IRPs
- Flush buffer IRPs
- System shutdown IRPs
- Various Fast I/O queries

The EFSD will not handle Directory Notification IRPs, nor will it support hard links (which are supported natively on NTFS on W2000 only): neither of these requests are required, and no expected user functionality will be lost without them. We are presently not supporting byte-locks; this may need to be revisited if the need arises.

In addition to the interfaces to the NT Executive, the EFSD will support various interfaces from other client components; all these will be sent via IOCTL calls. The first ones listed are simple support interfaces; the interfaces between the ECM and the EFSD follow these.

An IOCTL coming in to the kernel—via a DeviceIoControl() call—has the following parameters:

- IOCTL control code
- input buffer pointer
- input buffer size
- output buffer pointer
- output buffer size
- pointer to a 4-byte variable to receive the number of bytes written to the output buffer
- pointer to an OVERLAPPED structure for asynchronous operation (always should be NULL for EFSD)

All of the following interfaces are described in terms of the IOCTL buffers sent and received for each control code.

The following interfaces are called from the controlling client component (StartClient).

Starting and stopping the file system

The eStream FSD will be loaded into the kernel when a system is rebooted; i.e., it is always resident. If applications access files on this FSD via a drive letter, then the file system is implicitly turned off while a symbolic link for that drive letter is not present. Even when a drive letter symlink exists, the EFSD will not accept requests until the START IOCTL is sent.

These IOCTL control codes will be defined for starting and stopping the eStream FSD:

```
IOCTL_EFS_START_FS
IOCTL_EFS_STOP_FS
```

Starting the FSD

The input buffer for the START IOCTL should have the following:

- version id: 4-byte identifier for the client component
- debug flags: 4-byte value indicating the debug level to use

The output buffer for this IOCTL will be filled with the following:

- version id: 4-byte identifier for the EFSD version present
- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BAD_VERSION
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already started.

Stopping the FSD

The input buffer for the STOP IOCTL should have the following:

- force: 4-byte value
  - 0: shutdown only if no outstanding files are open
  - 1: shutdown regardless of state of open files

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_DUPLICATE_REQUEST
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

A DUPLICATE_REQUEST error is returned if the FSD is already stopped.

Cache management interfaces

The following two interfaces are defined for use by the ECM to potentially invalidate data in the NT File Cache.

These IOCTL control codes will be defined for cache management for the eStream FSD:

```
IOCTL_EFS_INVALIDATE_FILE
IOCTL_EFS_INVALIDATE_DIR_CONTENTS
```

Invalidating a file

The input buffer for the INVALIDATE_FILE IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open file that must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

If in fact the file is open, but not present in the NT File Cache, this IOCTL will simply succeed; no error is returned.

Invalidating directory contents

The input buffer for the INVALIDATE_DIR_CONTENTS IOCTL should have the following:

- handle: 4-byte EFSDFileHandle for the open directory whose contents must be invalidated

The output buffer for this IOCTL will be filled with the following:

- status: 4-byte value, with one of the following:

```
EFS_STATUS_SUCCESS
EFS_STATUS_BUFFER_TOO_SMALL
EFS_STATUS_FILE_NOT_OPEN
EFS_STATUS_ABNORMAL_TERMINATION
```

The status return value from this IOCTL will be one of the following:

- STATUS_SUCCESS
- STATUS_INVALID_DEVICE_REQUEST

General file system requests

All file system requests that cannot be completely handled by the EFSD will be passed on to the ECM. Since the ECM is likely to be a user-mode service, the EFSD cannot call it directly; thus these "calls" are made by having the ECM send IOCTLs to the EFSD to get and fulfill requests. Each file system request requires multiple IOCTLs sent from the ECM to the EFSD:

1. The ECM sends an IOCTL to the EFSD to get the next request
2. The ECM sends a second and/or third IOCTL to finish the request

The following IOCTL control codes will be defined by the EFSD for use by the ECM:

```
IOCTL_EFS_GET_REQUEST
IOCTL_EFS_RETRY_REQUEST
IOCTL_EFS_GET_CREATE_NAME
IOCTL_EFS_FINISH_CREATE
IOCTL_EFS_FINISH_CLOSE
IOCTL_EFS_FINISH_READ
IOCTL_EFS_GET_WRITE_DATA
IOCTL_EFS_FINISH_WRITE
IOCTL_EFS_GET_RENAME_TARGET
IOCTL_EFS_FINISH_RENAME
IOCTL_EFS_FINISH_DELETE
IOCTL_EFS_FINISH_METADATA_READ
IOCTL_EFS_FINISH_METADATA_WRITE
```

For the DeviceIoControl() call sending IOCTL_EFS_GET_REQUEST, these parameters are invariant:

- the IO control code will be IOCTL_EFS_GET_REQUEST
- input buffer pointer will be NULL
- input buffer size will be 0
- output buffer must be non-NULL
- output buffer size must be **at least 40 bytes**—this is the largest buffer needed for any request (subject, of course, to slight modifications)
- pointer to bytes returned will be non-NULL
- overlapped pointer will be NULL

The IOCTL_EFS_RETRY_REQUEST is sent by the ECM (or some other user-space client component) to tell the EFSD that, yes, it needs to delete all intermediate information about a request already sent back with a GET_REQUEST call, and put the request back on the list for the ECM to retrieve. This eases implementation issues for the ECM. The input buffer for a RETRY_REQUEST is:

- request id of the previously retrieved request

There is no output buffer for a retry request IOCTL.

What follows is a list of file system requests from the NT I/O Manager, and the IOCTL calls needed from the ECM to service those requests. For all cases, if the EFSD writes to the output buffer for an IOCTL, the "bytes returned" field is written with the number of bytes written.

Create

This is used for both create and open, for files and directories.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Create request
- request id: a 4 byte field that will be subsequently sent in the calls to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- flags: 4 bytes, one or more of the following ORed together
    - CREATE_ONLY: fail if file exists already
    - OPEN_ONLY: fail if file does not exist already
    - TRUNCATE: overwrite existing file
    - DIRECTORY: create a directory
    - FILE: create a plain file
    - DELETE_ON_CLOSE: delete file on last close
    - IGNORE_CASE: obvious
- permissions: 4 bytes, one or more of the following ORed together
    - READ
    - WRITE
    - EXECUTE
- length of filename: 4 bytes, specifying the byte size needed for the Unicode string sent in the next call

Total size of output buffer: 24 bytes

GET_CREATE_NAME

The input buffer for this IOCTL should have the following data:

- request id: the id sent in the previous call

The output buffer for this IOCTL will be filled with the following information:

- request id for this transaction
- fully qualified name as a counted Unicode string (including drive letter, if any): the length needed was sent back in the GET_REQUEST call

FINISH_CREATE

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- handle: the 4-byte handle for this opened file, that can be used for subsequent file system requests. A unique value will indicate a bad handle, and a failed Create
- a Metadata buffer: the metadata for the created/opened file/directory.

The output buffer for this IOCTL should be NULL.

Note that a TRUNCATE Create request should cause the metadata sent back to reflect the possibly new (zero) length.

Close

This closes a handle of a previously opened file or directory. The EFSD will optionally send the updated metadata for this file in the GET_REQUEST output buffer. If the file has been modified in any way, the metadata fields will be non-zero; else they will all be zero.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Close request
- request id: 4 bytes, for use in subsequent calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for the previously opened file
- metadata for this file/directory: 24 bytes
    - creation time stamp
    - modification time stamp
    - file/directory size in bytes
    - attributes (as described above)

Total size of output buffer: 40 bytes

FINISH_CLOSE

The input buffer for this call should contain the following:

- request id for this transaction
- status: the NTSTATUS for this request

Read

This is used for reading file data.

GET_REQUEST

The output buffer for the IOCTL_EFS_GET_REQUEST will be filled with the following:

- type: a 4 byte field that indicates a Read request
- request id: a 4 byte field that will be subsequently sent in the IOCTL to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to read from
- length: 4 bytes, the length of the read, in bytes

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

Total size of output buffer: 24 bytes.

FINISH_READ

The input buffer for this call should have the input buffer filled as follows:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- the number of bytes successfully read; 0 on error
- the data from the read; not present on error

The output buffer for this IOCTL should be NULL.

Write

This is used for writing file data.

GET_REQUEST

The output buffer for this will be filled with the following:

- type: a 4 byte field that indicates a Write request
- request id: a 4 byte field that will be subsequently sent in matching calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes, the handle for this previously opened file
- offset: 4 bytes, the file offset, in bytes, to write to
- length: 4 bytes, the length of the write, in bytes
- file length: 4 bytes, the length the file will be **if** this write succeeds

Total size of output buffer: 28 bytes.

NOTE: The buffer requested in the (offset, length) pair will *not* cross a 4K page boundary.

GET_WRITE_DATA

This IOCTL will have an input buffer with:

- request id for this transaction
- status: if not STATUS_SUCCESS, this ends the request; the output buffer is untouched, and no FINISH_WRITE call is expected.

And the output buffer will be filled with:

- request id
- data buffer for the write—the byte length sent in the previous GET_REQUEST

FINISH_WRITE

For this finishing request IOCTL, the input buffer has these contents:

- request id: the matching id from the GET_REQUEST call
- status: the NTSTATUS result from this request
- bytes actually written; should be equal to requested bytes unless failure occurs.

Rename

This is used for renaming a file or directory.

GET_REQUEST

The output buffer for this IOCTL will be filled with the following:

- type: a 4 byte field that indicates a Rename request
- request id: a 4 byte field that will be subsequently sent in the FINISH_REQUEST call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- length of target name: 4 bytes; the byte length needed for a counted Unicode string for the target name

Total size of output buffer: 20 bytes.

GET_RENAME_TARGET

The input buffer for this call will have the following:

- request id for this transaction
- status: if not STATUS_SUCCESS, then this terminates the request: the output buffer is not touched, and no FINISH_RENAME call should be sent

The output buffer will be filled with the following:

- request id
- target name: a counted Unicode string, using the same number of bytes as sent in the GET_REQUEST output buffer

FINISH_RENAME

The input buffer for this call should have the following:

- request id
- status: NTSTATUS for the transaction

Delete

This is used for deleting a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Delete request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_DELETE

The output buffer for this call should be NULL.

The input buffer should have the following contents:

- request id: matching id from the GET_REQUEST call
- status: NTSTATUS of this request

Metadata read

This is used for requesting metadata about a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata request
- request id: a 4 byte field that will be subsequently sent in the call to match this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory

Total size of output buffer: 16 bytes.

FINISH_METADATA_READ

The output buffer for this IOCTL will be NULL.

The input buffer should have the following contents:

- request id: id from the corresponding GET_REQUEST
- status: NTSTATUS for this operation
- the following data about the file or directory:
  - creation time stamp
  - modification time stamp
  - file/directory size in bytes
  - attributes (as described above)

Metadata write

This is used for setting metadata for a file or directory.

GET_REQUEST

The output buffer for this call will be filled with the following:

- type: a 4 byte field that indicates a Metadata Write request
- request id: a 4 byte field that will be subsequently used for all calls for this request
- retry count: 4 bytes—how many retries this GET_REQUEST corresponds to. First time, this is 0.
- handle: 4 bytes; the handle for this previously opened file or directory
- metadata for this file/directory: 24 bytes
  - creation time stamp
  - modification time stamp

- o file/directory size in bytes
- o attributes (as described above)

Total size of output buffer: 40 bytes.

FINISH_METADATA_WRITE

The input buffer should have the following contents:

- request id: from the previous call
- status: NTSTATUS for this request

Component design

This section is organized in the following manner:

1. General layout of the eStream file system driver
2. General observations about the low level design
3. Organization of data structures
4. Description of the algorithms for communication with the ECM
5. Description of each dispatch routine

Layout

The EFSD will be generally organized in the following manner:

- All major IRPs will have their own dispatch routine.
- All actual I/O requests to the ECM will be generalized from the dispatch routines to a set of routines that handle the communication with the ECM, to isolate this aspect.
- All utility functions will be in their own file or files.

General points

The design of the EFSD will look a lot like the sample FSD from Rajeev Nagar's NT FS Internals book, which looks a whole lot like the Fastfat FSD source from the NT IFS kit.

Here is a list of general points that can be made about the EFSD:

- Any IRP that can be handled asynchronously will be posted to a work queue; this means that the dispatch routine for such an IRP must be able to handle being called in a context other than the original requestor.
- There are no volumes, and no Volume Parameter Block or Volume Control Block. There isn't a VPB for a network redirector; I've verified this with the

LanManager redirector. Hence we don't have to support any operations on a volume in EFS.

- We will not allow the creation of paging files in EFS. There is a bit available for a Create IRP that specifies this, and we can complete the IRP with an unimplemented error return code.
- All file synchronization will be on a File Control Block (FCB) basis, using the standard Resource and PagingIoResource ERESOURCE objects used by the rest of the Windows Executive.
    - User requests will be synchronized by acquiring the main Resource—shared for reads, shared for most writes, and exclusive for file size changes, deletion, etc.
    - Paging I/O requests will be synchronized by acquiring the PagingIoResource—again, shared for reads, shared for most writes. Exclusive access will be needed to set file sizes.
- Most disk file systems have a resource associated with a VCB, which is acquired exclusively for creation/deletion etc. We will have a global EFS resource for this, since there are no VCBs.
- Asynchronous requests will be handled by posting the IRP to the CriticalWorkQueue, and marking the IRP as pending.
    - A common worker routine will be used for all async posts, which will dispatch the IRP to the appropriate real IRP routine when it's invoked.
    - An async request will be defined as one that IoIsOperationSynchronous() returns false, and the EFSD is the top-level component (see below)
- The EFSD will track the top-level IRP for the thread whose context it is running in. In particular,
    - No async processing request will be honored unless the EFSD is the top-level component
    - No cache manager requests will be made unless the EFSD is the top-level component
    - EndOfFile size—that is, the true size of the file—will not be extended or changed by paging I/O
- EFS will not support holes in files, and hence the ValidDataLength FCB field will be set to disable this. This means the AllocationSize for an eStream file/directory will always be equal to the EOF size.
- Most fast I/O routines will be supported in EFS. We will use the FSRTL supplied routines for fast reads and writes.
- All cache manager resource acquire/release callbacks will be supported. All will point to common routines that simply acquire or release the main or paging I/O resource for the FCB. The Context pointer passed into all of them will be the FCB for the stream.
- Synchronous read/write requests will update the CurrentByteOffset in the File object.
- Each Create will result in a unique Context Control Block (CCB) data structure; this will be small, and only hold those few fields needed:
    - For the Directory Control IRP, a CCB needs to hold the current entry index and the pattern originally used—for subsequent queries

  - A field for various flags
- A single FCB will represent all current open instances of a file. When a Create request comes in, the EFSD will search the current open FCBs to try to find one matching this file/directory name.
  - For now, this will be a hash table on the file name. We can improve this as needed.
  - The EFS global resource must be acquired exclusively:
    - before the global FCB data structure is searched. **Why? If it's just a read, can't we acquire it non-exclusively?**
    - before a new FCB is added to the list
    - before an FCB is deleted from the list
- EFS will not support open by file ID; hence the FileInternalInformation class for a File Information IRP will not be supported.
- Actual I/O will be directed to standard routines in a separate file, so they can be isolated and updated easily as our method of transferring data changes.
- Here's how to do file/directory renames:
  - The I/O manager will send to the EFSD this sequence:
    - Create for source
    - Create for target, with the SL_OPEN_TARGET_DIRECTORY flag set
    - Set Information with a Rename request for the source, sending the target directory FileObject handle and the target name in the FileInformationClass record.
  - EFSD needs to do this:
    - When it receives the Create for the target and the target *directory* exists, return STATUS_SUCCESS, and change the name in the FileObject to the basename of the target (the full pathname of the target is sent in), and set the Status.Information to FILE EXISTS or FILE DOES NOT EXIST, as appropriate. If the target directory doesn't even exist, return PATH NOT FOUND.
    - When it receives the Set Info request, if all the flags check out (e.g., if the file exists, ReplaceExisting must be TRUE), send a Rename request to the ECM.
- Reads and writes to only regular files will be supported, not to directories.
- Any code that touches user buffers or can call routines that may throw exceptions must be guarded by a try/except block.
- Some tips on memory allocation (from /perforce-doc-dir/osrdocs/defensive-driv.html)
  - We will use our own memory allocation/deallocation routines, instead of ExAllocatePool() et al. directly
  - These routines can do various checks for trashing memory:
    - fill allocated memory with a pre-defined bit pattern, instead of zeroes; fill deallocated memory with a different pattern.
    - allocate a header/trailer with standard information, like where allocated, from what pool, etc.

    - change the bit pattern in the header/trailer on deallocation, and look for freeing memory twice
  - We probably want to allocate using lookaside lists, since we'll be allocating and deallocating smallish chunks of memory for our data structures.

Data structures

The following are major data structures used internally by the EFSD. Data structures used to communicate with the ECM are described in the following section.

NodeIdentifier

A NodeIdentifier is a simple structure that starts all other structures used in the EFSD. This makes a good debugging check to insure that we receive and are operating on the right type of data. It consists of two fields: an identifier field, and the total structure size.

FCB

The FCB is a critical data structure for the file system driver. There is one FCB structure allocated for each unique file or directory that is currently open—regardless of how many open handles there are for this entry. Multiple CCB structures can point to a single FCB.

Logically at least, part of an FCB is exposed to the Cache Manager and VMM to support caching and paging I/O. We will follow the example of Rajeev Nagar's book, and embed the FSRTL_COMMON_FCB_HEADER and other required structures directly in an FCB.

Here are the basic contents of an EFSD FCB:

- The required FCB contents above
- An open handle count: incremented on Create, decremented on Cleanup
- A reference count: incremented on Create, decremented on Close. The semantics of NT file requests require these two be used together to determine when an FCB can be deleted
- A pointer to the next FCB on its hash list
- A pointer to the first CCB opened for this FCB
- The fully qualified name of the file/directory
- The Metadata associated with this file/directory
- A SHARE_ACCESS structure used to check sharing violations
- Various flag bits

CCB

A CCB represents a currently opened handle to a file or directory. If two processes have the same file opened, there will be a unique CCB allocated for each process.

Here are the basic contents of an EFSD CCB:

- A pointer to its corresponding FCB
- A pointer to the next CCB opened for its FCB
- A pointer to the FileObject opened for this file/directory
- The current file index for a directory query
- The directory search pattern used for directory queries for this opened handle
- Various flag bits needed

IrpContext

An IrpContext structure encapsulates the interesting data from an IRP, and the current stack location, for easy access. This structure is allocated on entry to dispatch routines, and used during processing, before being deallocated on exit.

Communicating with the ECM

I tend to divide a file system driver into two logical parts:

1. A front-end that understands the NT FSD interfaces and semantics
2. A back-end that actually perform the requested actions

Our back-end is the (admittedly ugly) interface for communicating with the ECM, which currently sits in user-space. It's very important to design the ECM such that its front-end and back-end are nicely separated: since the ECM may move to kernel space, or we might find better interfaces for them to communicate with each other, we need to design with this in mind.

Given the current ECM interfaces defined above, here is a basic design to handle them:

- An EfsdRequest object will be created for each request that must be handled by the ECM.
- Each dispatch routine that results in a request to the ECM will allocate an EfsdRequest and send it to a common routine for further processing.
- New requests will be placed in a NewRequest queue.
- Requests that have been "sent" to the ECM, but not yet finished, will be removed from the NewRequest queue and placed in a PendingRequest list.
- Finishing a request entails removing it from the PendingRequest list, returning the contents to the dispatch routine, and destroying the request object.

Data structures

EfsdRequest

This contains:

- request id: a number that uniquely identifies this request
- type: the type of request (e.g., Create, Write)
- FCB pointer for the file/directory for this request
- IrpContext pointer for this request
- a kernel event object used for signaling that the request is satisfied

NewRequestSemaphore

The EFSD device object's device extension will contain a semaphore dispatcher object, initialized to non-signaled, and with an initial limit of MAX_LONG. When a request is added to the NewRequest queue, this semaphore is released (and the count is incremented by 1); the GET_REQUEST IOCTL will wait on this semaphore object (which decrements the count by 1).

NewRequestQueue

This is actually a kernel-managed interlocked list that is allocated in the device extension area, guarded by a spin lock that's also located in the device extension. Requests will be added to the tail, retrieved from the head.

PendingRequestList

This list must be searched when an ECM non-GET_REQUEST IOCTL is received, so we can't use an interlocked list. We'll use a global single-linked list structure, with elements allocated from a dedicated lookaside list using non-paged memory. Before a thread can access the list elements, it must acquire a mutex.

Algorithm

- All dispatch routines, and asynchronous read/write routines, will call LowLevelPostRequest() to have their requests satisfied. LowLevelPostRequest() is itself synchronous; that is, the code calling it is something like this:

```
status = LowLevelPostRequest(fcb, irp_context);
     // we're done;
     // do all deallocation and cleanup needed
IoCompleteRequest(irp, ...);
```

- LowLevelPostRequest() will do the following:

```
if this is a read or write IrpContext
    see how many requests are needed to satisfy the IRP: can't span page boundary
    allocate the N requests
    allocate an array of N event pointers to hold the request events
    assign the pointers to the array
    if > THREAD_WAIT_OBJECTS
        allocate an array of N PKWAIT_BLOCKS
```

```
else if this is a directory query IrpContext
    look at the FileSize of the directory FCB
    allocate enough read requests to read all directory data from the ECM
    as above, allocate an array of event pointers, wait objects (if necessary)
else
    allocate a new EfsdRequest for the incoming FCB and IrpContext
place all requests on the NewRequestQueue, using ExInterlockedInsertTailList()
call KeReleaseSemaphore() on the NewRequestSemaphore
if multiple requests generated
    call KeWaitForMultipleObjects() on the request object's event
else
    do a KeWaitForSingleObject() on the request object's event
when the event(s) is/are signaled
    fill in the values into the IRP
    deallocate the EfsdRequest(s)
    return status
```

- When a GET_REQUEST IOCTL comes in from the ECM, the EFSD will do the following:

```
do a KeWaitForSingleObject() on the NewRequestSemaphore
remove the first request from the NewRequestQueue,
                using ExInterlockedRemoveHeadList()
lock the PendingRequestList mutex
place this request on this list
release the mutex
fill in the IOCTL output buffer identifying the request
complete the IOCTL IRP
```

- When a RETRY_REQUEST IOCTL is received, the following takes place:

```
lock the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
enqueue the request on the NewRequestQueue—on the list head
release the NewRequestSemaphore
complete the IOCTL IRP
```

- When any "finishing" IOCTL is received—i.e., the second of two or the third of three—the following is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
remove the request from the list
release the mutex
do all buffer copying, set all flags,
            and otherwise insure the input IRP has the correct state
signal the EfsdRequest event
complete the IOCTL IRP
```

- When any other request IOCTL is received—e.g., the second of three—this is done:

```
acquire the PendingRequestList mutex
search the list by request id; error if not found
release the mutex
fill in the output buffer of the IOCTL as appropriate
complete the IOCTL IRP
```

Dispatch routines

DriverEntry

This does a whole slew of initialization, including the dispatch table, fast I/O table, the cache callbacks, the FCB hash table and its synchronization object, creates the FS device object, and sets up the interface with the ECM.

Create

There is one Create routine; there will be no async processing of Create requests. Ultimately, its job is to send a create request to the ECM, and return SUCCESS or not to its caller. Here is a general algorithm for this routine.

```
create an IrpContext
if a page file is requested
    return error
generate the absolute pathname of the requested file:
    if OPEN_TARGET_DIRECTORY specified
        if OPEN_ONLY not specified
            return error
        generate the pathname of the parent directory of the requested file
    if a related file object is specified
        if the related file is not a directory
            return error
        if the related filename doesn't start with '/'
            return error
        if the input filename starts with '/'
            return error
        concatenate the input filename with the related file directory
    else use the input filename
        if the input filename does not start with '/'
            return error
acquire the global EFS resource exclusively
search the FCB hash table for this file by name
if not found
    call LowLevelPostRequest() to send the request to the ECM
    if error is returned from ECM
        return the correct error: FILE NOT FOUND or PATH NOT FOUND
    create a new FCB and add to the hash table
    call IoSetShareAccess() for this FCB
else
    check input attributes/flags against those in FCB
        if opening for write or delete on close
            call MmFlushImageSection()
            if this fails
```

```
                return error
        if failing mismatch—e.g., if IoCheckShareAccess() fails
            return error
    create a new CCB for this file
    set all appropriate flags on CCB and/or FCB
        COMMON_FCB_HEADER flag is set to FastIoIsPossible
    set all fields on input FileObject:
        write through flag
        FsContext points to common FCB header
        FsContext2 points to CCB
    if OPEN_TARGET_DIRECTORY is specified
        search for the input target object:
            look for this in the FCB hash table
            if not found
                send a Create request for this file to the ECM
            if this target filename exists
            set Status.Information to FILE_EXISTS
        else
            set Status.Information to FILE_DOES_NOT_EXIST
        if a Create was sent to the ECM for the input FileObject
            send a Close request for the input target to the ECM
        change the name in the FileObject to the basename of this file
        the CCB and FCB remain opened for the target directory
    all necessary data structures should be deallocated, for success and error
    release the global EFS resource
```

Cleanup

No async posting of Cleanup requests will be done. Algorithm:

```
    acquire the global EFS resource, and the FCB Resource, for this file, exclusively
    if this file is marked for deletion
        if this is the last open handle for the file
            acquire the PagingIoResource exclusively
            set the file size in the FCB to 0
            release the PagingIoResource
            purge the cache, if necessary, with MmFlushImageSection()
            call LowLevelPostRequest() to send a Delete request to the ECM
    decrement the count of open handles in the FCB
    if caching is on
        flush the cache by calling CcUnitializeCacheMaps()
    any time stamps must be updated if accesses were done using fast I/O
    set the FO_CLEANUP_COMPLETE flag in the FileObject
    call IoRemoveShareAccess()
    release the global EFS and FCB Resources
```

Close

There will be no async posting of Close requests. Note that we only send a Close request to the ECM if this is the last close for an open file—i.e., we're matching Close requests with Create requests.

Algorithm:

```
acquire the global EFS resource exclusively and the FCB Resource
deallocate the CCB
decrement the reference count for the FCB
if the FCB ref count is now 0
    remove the FCB from the hash table
    deallocate the FCB
    call LowLevelPostRequest() to send a Close request to the ECM,
                                        updating metadata if necessary
release the global EFS resource and the FCB Resource
```

Read

Reads will definitely be open to async posting. A general algorithm:

```
if the read length is 0
    return success
if the target file object is a directory
    return error

if this IRP can be handled async
    post for async processing

if this read is non-buffered, and it's not for paging I/O, and
                        there is a mapped data section object for this file
    acquire the FCB main Resource exclusive, and the PagingIoResource shared
    call CcCacheFlush() on the range of this read (current byte offset, length)
    release the FCB resources
if this is for paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared
if this read starts beyond EOF
    return EOF
if the read length goes beyond EOF
    truncate the length
if this is a buffered read
    if the PrivateCacheMap is NULL
        call CcInitializeCacheMap()
    if this is an MDL read
        call CcReadMdl()
    else
        call CcCopyRead(), using either an allocated MDL or the UserBuffer

else (it's non-buffered)
    call LowLevelPostRequest() to send a Read request to the ECM

if this is a synchronous, non-paging read
    update the current byte offset in the FileObject
set the number of bytes read in the Status.Information field
release any acquired resource and deallocate appropriate data structures
```

Write

Writes will definitely be open to async posting. A general algorithm:

```
if the write length is 0
    return success
if the input file is a directory
    return error
if the file not opened with write permissions
    return error

if this IRP can be processed asynchronously
    post for async processing

if this is a buffered write
    call CcCanIWrite()
    if false
        we have a hard error; fail
if this is paging I/O
    acquire the PagingIoResource shared
else
    acquire the main Resource shared.
if the length is
        (Low == FILE_WRITE_TO_END_OF_FILE) && (High == 0xffffffff)
    we're to write at EOF
if this is a non-paging, non-buffered write, and
                            there is a mapped data section object for this file
    acquire the global EFS resource exclusive
    acquire the PagingIoResource shared, starving exclusive waiters
    call CcCacheFlush() on the range for this write
    release the PagingIoResource
    return error if the cache flush failed
    acquire and release the PagingIoResource exclusive (to serialize)
    call CcPurgeCacheSection() on the range for this write
    release the global EFS resource

if this is a paging I/O write
    if the starting offset is beyond the current EOF
        return success
    if the ending offset is beyond the current EOF
        truncate the write length to EOF

if this is a buffered write
    if the private cache map is NULL
        call CcInitializeCacheMap() for this file
    if the write will extend the file size
        release the resource acquired
        re-acquire the resource exclusive
        call CcSetFileSizes() inform the cache manager
    if this is an MDL write
        call CcPrepareMdlWrite()
    else
        call CcCopyWrite(), using either an allocated MDL or the UserBuffer

else
    call LowLevelPostRequest() to send the write request to the ECM

set the number of bytes written in the Status.Information field
update the file size fields in the FCB if this write extends the length
if this is a non-paging write
```

```
        set the CCB modification flag
        if this write is synchronous
            update the CurrentByteOffset field in the FileObject
    release any acquired resource and deallocate appropriate data structures
```

Fast I/O Read

Initially at least, we'll just set the fast I/O read routine to FsRtlCopyRead().

Fast I/O Write

Initially at least, we'll just set the fast I/O write routine to FsRtlCopyWrite().

Fast I/O Query Basic Info

This will just fill in the basic info buffer with the data in the FCB.

Fast I/O Query Standard Info

This will just fill in the standard info buffer with the data in the FCB.

Fast I/O Query Open

This will just fill in the network open info buffer with the data in the FCB, if the file exists. Some empirical observations I've made using NTFS:

- Regardless of whether the file exists or not, this will return TRUE (all fast I/O routines are boolean)
- If the file does not exist, it will set the EOF size in the buffer to 0. The AllocationSize must be non-zero. All other fields seem to be don't cares.
- If the file exists but is zero length, then both EOF and AllocationSize will be 0.
- The IRP sent to this routine is for an IRP_MJ_CREATE; we can use more than just the name to identify the file, but also the security characteristics or whatever else is sent in the IRP.

File Query Info

Standard queries will be supported; these however will not:

- FileInternalInformation—no OPEN_BY_FILE_ID
- FileEaInformation—no EA data
- FileCompressionInformation—no on-disk compression
- FileStreamInformation—no multiple streams

File Set Info

These actions will be supported:

- EndOfFile size changes
- AllocationSize changes
- Time stamp changes
- File position changes
- File disposition changes—delete pending
- File rename requests

Here is a general algorithm:

```
if AllocationSize or EOF size change
    if the new size is smaller than the current size
        call MmCanFileBeTruncated(), to ask the VMM if this is okay
        if yes
            call CcSetFileSizes()
        else
            return STATUS_USER_MAPPED_FILE error
    else
        send a Metadata Set request to the ECM with new, extended size
        if status returned is error
            return error (disk space full)
    update AllocationSize and FileSize fields of required FCB header

else if time stamp change
    send Metadata Set request to ECM
    if error returned
        return with error

else if file position change
    update FileObject CurrentByteOffset field

else if disposition (delete) change
    if the Delete flag in the IRP not set
        clear delete on close FCB flag
    if file already marked for delete on close
        return success
    if file not open with write permission
        return error
    if MmFlushImageSection() fails
        return error
    if this is a directory, and the directory is not empty
        return error
    set delete on close flag in FCB

else if rename change
    if the source name is for a directory
        if the directory has any open files/directories under it
            return error

    if Parameters.SetFile.FileObject is NULL (simple rename)
        target dirname is the same as dirname of IRP FileObject
        target basename is Buffer.FileName

    else
        if Buffer.RootDirectory is NULL (fully-qualified rename)
```

```
            fully qualified target name is Buffer.FileName

        else  (relative rename)
            call ObReferenceObjectByHandle() to get the file object of the relative directory
            from file object, get (fully qualified) dirname of target
            append Buffer.Filename to root directory dirname to get fully qualified target

    if the target name isn't on EFS
        return error
    if target exists
        if Parameters.SetFile.ReplaceIfExists is FALSE
            return error
        if target is a directory
            return error
        if target is read-only
            return error
        if target has any open handles to it
            return error

    call LowLevelPostRequest() to send a Rename request to the ECM
```

Directory Query

Directory queries turn into directory read requests to the ECM. The EFSD will take the contents of the read buffers and fill the IRP_MN_QUERY_DIRECTORY buffers sent to it from the NT Executive by parsing the directory contents.

Note: this design does not use the NT Cache Manager for metadata or for directory entries, nor does the EFSD store the directory contents anywhere in its data structures. It will always go back to the ECM for directory queries. Given that most directory queries occur in this order:

Create
Directory query
Close

unless we can cache the contents in a location more persistent than an FCB, we will need to resubmit the request to the ECM for each new directory query. If this poses a performance problem, we need to handle it then.

Directory queries are subject to async processing.

This is an ugly NT interface. Here are some general points regarding directory queries, and how they will be implemented on EFS:

- These requests come in from the I/O Manager in a context-sensitive sequence. I.e., a request will come for the initial N directory entries; the next request will be for the next M entries; etc. Kind of like strtok().
- Thus, state must be maintained from request to request. This state will be kept in the CCB for a file stream, and consists of:

    - Pattern sent in on first request
    - Index of n'th entry to start retrieving with
- My experience is that the INDEX_SPECIFIED flag is never set in a directory control query, even on queries subsequent to the initial one.
- For EFSD, the FileIndex will represent the offset of the fixed-length portion of a directory entry as returned by the ECM.
- Initially, at least, all directory queries will cause the EFSD to read all the entries for that directory from the ECM. We can modify this later if needed.

Here is a general algorithm, based on the sources for the Fastfat driver:

```
if the FileObject is not for a directory
    return error

post this for async handling, to read all directory pages from the ECM

if the CCB pattern field is empty and the CCB flag "match all" isn't set
    this is the initial query
    acquire the FCB Resource exclusive
else
    acquire the FCB Resource shared

get a pointer to the input buffer, using either an MDL or the UserBuffer
if this is the initial query
    parse the FileName pattern
    save the pattern in the CCB
    if the pattern is "*"
        set the "match all" flag in the CCB

if SL_RESTART_SCAN is specified
    use an index of 0 for the query
else if SL_INDEX_SPECIFIED is specified
    use the input index for the query
else
    if this is the initial query
        use an index of 0
    else
        use the index saved in the CCB

start with the directory entry corresponding to the starting index
if this index is beyond the number of entries in the directory, and this is not the initial query
    return STATUS_NO_MORE_FILES
while there are more directory entries
    if the directory entry name matches the pattern in the CCB,
                                        using FsRtlIsNameInExpression()
        if there isn't room in the buffer for this entry
            break
        write the entry in the input buffer
        if there is a next directory entry beyond the current one
            the FileIndex field is set to the offset of this next directory entry
        else
            the FileIndex field is set to 0
        if there is a previously written entry
            fix up the NextEntryOffset in the previous entry to the byte offset of this entry
```

```
            if SL_RETURN_SINGLE_ENTRY specified
                break
            8-byte align the current pointer in the user buffer
        advance to next directory entry

    if we wrote nothing
        if we stopped because of no room
            return STATUS_BUFFER_OVERFLOW
        else
            return STATUS_NO_SUCH_FILE

    update the index field in the CCB
    Status.Information is set to the number of bytes written
    return STATUS_SUCCESS
```

File System Query Info

Empirically, I've noticed that the LanMan redirector returns failure for most of these requests. So, except for any user-defined FSCTL requests we want to define, I'm going to fail all of these until it turns out we need to do otherwise.

File System Set Info

Ditto for this IRP type too.

Volume Query Info

We at least need to minimally implement these requests:

- FileFsAttributeInformation
- FileFsVolumeInformation
- FileFsDeviceInformation

This will be handled solely by the EFSD; the request will not go out to the ECM. File system size requests will not be handled.

Volume Set Info

We will fail all requests of this type.

Flush Buffers

A buffer flush request for a file stream will mean the following:

- If the file stream isn't buffered, return immediately
- The FCB main Resource is acquired exclusive
- The Cache Manager is told to flush the buffer for the byte range of the file
- The resource is released

A buffer flush for a directory is a successful NOP.

The buffer flush request will insure that contents in the NT File Cache are written to the ECM (as a normal write request); the buffer flush request itself will not be propagated to the ECM.

Testing design

Unit testing plans

The EFSD will be tested apart from integrating with the ECM or the rest of the eStream client. Some points:

- There will be a (relatively) simple stand-in user process for the ECM, to get requests from the EFSD and handle them locally.
- As much EFSD functionality as possible will be done using user-mode test cases (e.g., open files, read/write files, delete files, etc.).
- Some functionality may need to be unit tested using another kernel-mode driver to send explicit IRPs to the EFSD.
- Filemon will be used to monitor the requests being handled by the EFSD.

Stress testing plans

I've heard of a file system filter driver test package available from Microsoft. This is probably the best way we have of stress testing the EFSD.

Coverage testing plans

We'll try to measure coverage on the EFSD. If there is a general kernel-mode method for measuring coverage that's used company-wide, we'll exploit it. Otherwise, there will be some primitive self-coverage instrumentation conditionally inserted in the driver code that we can use at least for major code paths.

Cross-component testing plans

There's not much to do here apart from normal interactions with the ECM.

Upgrading/Supportability/Deployment design

For supportability, there will be solid debugging aids—using printfs—built into the EFSD sources. Additionally, aside from good error codes returned from its interfaces, the EFSD will explore diagnostic traces optionally dumped for deployment.

Issues with stakes in the ground

- At present, I am assuming that there is a single drive letter associated with the EFSD, though there's no technical reason why this must be so. If indeed we organize the client system and the eStream file hierarchies to have multiple drive letters, either the EFSD or the ECM will need to parse the drive letter and do the right thing.
- This design assumes that 8.3 DOS-style filenames need not be supported. Adding such support will increase the complexity of the EFSD, as well as many other components in the eStream system: on the client, on various servers, and on the content builder.
- No support is provided in this design for:
    a. byte locks
    b. directory notification
    c. file open by file id
    d. file system control requests

Open Issues

1. I'm unclear on how to use the NT File Cache for metadata and directory contents. For now, I'm ignoring such matters, and we will only be caching file contents.
2. I do not know how to hook up the EFSD to UNC names. That is, I don't know how to set things up to have all file accesses like \\ASP1\Office\winword.exe be directed to the EFSD by the NT I/O Manager.
3. This design doesn't cover exactly how IRPs are posted for asynchronous processing. The SFSD example in Rajeev Nagar's book really isn't sufficient for some of what we need to do. Also, it's unclear to me what value there is to returning STATUS_PENDING and handling a request asynchronously if the caller is blocking anyway.

Estream 1.0 Functional Specification * Use Cases * Revision 0.4 * [redacted]

EStream Client Functionality:

- ⇨ Installation of eStream client code
    - o Use browser to contact ASP Web Server, download bits to be installed.
    - o Install z: file system hooks & setup to have z: mounted at system boot.
    - o Install eStream client code, which services z: file sys requests from local cache or from servers & which handles sideband communication w/ servers, and setup to activate estream client code at system boot.
    - o Install NoCluster.sys to disable page fault clustering at system boot.
    - o Install eStream browser plug-in, which can receive messages from ASP Per-User Account Server alerting eStream client when new app purchased. [Sending unsolicited messages may not be possible thru firewall.]
- ⇨ Execution of eStream client code
    - o Respond to z: file sys requests. For apps w/ active online connection(s), user sees the detailed contents on z: that one would see if one had installed the apps locally, though copy access may be controlled. For apps to which the user has obtained offline access, user also sees the appropriate detailed contents on z: (although the files are actually in eStream client-managed memory on local disk). For each app whose connection is currently inactive, user sees a placeholder file entry on the z: file system (on which the user can double-click to launch an active connection).
    - o Establish/terminate session logins to ASP Per-User Account Server, upon user request or upon receiving app purchase message from browser plugin.
    - o Obtain/cache unique certificates for purchased applications from ASP Per-User Account Server.
- ⇨ Register with ASP
    - o Use browser to contact ASP Web Server.
    - o Follow ASP process to register.
    - o User obtains login/password, used by estream client code for sessions.
    - o ASP records user's login/password on ASP Per-User Account Server.
- ⇨ Purchase of application
    - o Use browser to contact ASP Web Server.
    - o Follow ASP process to buy app; user is given unique certificate for app.
    - o App purchase & certificate recorded on ASP Per-User Account Server.
    - o User is directed to go to client & request app installation and/or ASP Per-User Account Server attempts to send message to eStream browser plug-in on user's preferred client system (if any), so client can begin app install.
- ⇨ Installation of application
    - o Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & a session id.
    - o Contact designated App Server using id info, download meta-data about app, potentially including registry/DLL/filesys spoofing info, prefetching info, initial cache contents for app. For offline installation, obtain all files.
    - o Perform initial installation & setup for app, after checking system for previously installed version of app & issuing any appropriate warnings.

- ⇨ Execution of application
    - o Send unique certificate for application to appropriate ASP DRM Server, get back id for closest/best App Server & session id.
    - o Contact designated App Server using id info, request file system data as necessary. Respond to running application's requests, collect usage data. Cache portions of application, file system info, & user preference info.
    - o Detect server connection issues (apparent loss of connection or connection response below acceptable threshold); negotiate with ASP DRM Server for alternative connection if need be.
    - o At exit from application (or at other selected times), save portions of cache to client nonvolatile memory. Upload usage information to ASP Per-User Account Server.
- ⇨ Uninstallation of application
    - o Remove all registry/DLL/filesys changes associated with app installation.
    - o Remove all meta-data about app.
- ⇨ Uninstallation of eStream client code
    - o Remove z: file system hooks, eStream client code, & nocluster.sys.

EStream Server Functionality in terms of kinds of eStream Servers responding to Clients [may be embodied in any number of physical server computer systems]:

1. App server
    - o functionally read-only
    - o serves .exes, .dlls, etc.
    - o contains install info (aka, eStream sets)
2. ASP web server
    - o used to get eStream client bits
    - o eStream browser plug-in
    - o handle other user queries, e.g., concerning available apps, current billing status
3. Per-user account server
    - o registration info, issue serial numbers for purchased apps
    - o accept/store uploaded info about app usage
    - o perhaps: user preferences for each app
4. DRM server
    - o authentification of users
    - o validate app licenses, track outstanding offline licenses
    - o hands out licenses for #1 above

Estream Server Management/Maintenance Functionality
- ⇨ Install/maintain eStream apps [aka Builder]
    - o Provide tool/methods to generate initial meta-data about app, including registry/DLL/file spoofing info, initial prefetching info, initial cache contents, etc.
    - o Provide tool/methods to place app & meta-data into public access area and to remove from public access areas
    - o Update meta-data as appropriate to reflect uploaded client usage info
- ⇨ Handle server traffic
    - o Support trouble-shooting of performance or correctness problems

- Perform automated load balancing
- Support online addition/reconfiguration of servers

⇨ Provide tools to process uploaded app usage info.

Open functionality questions:

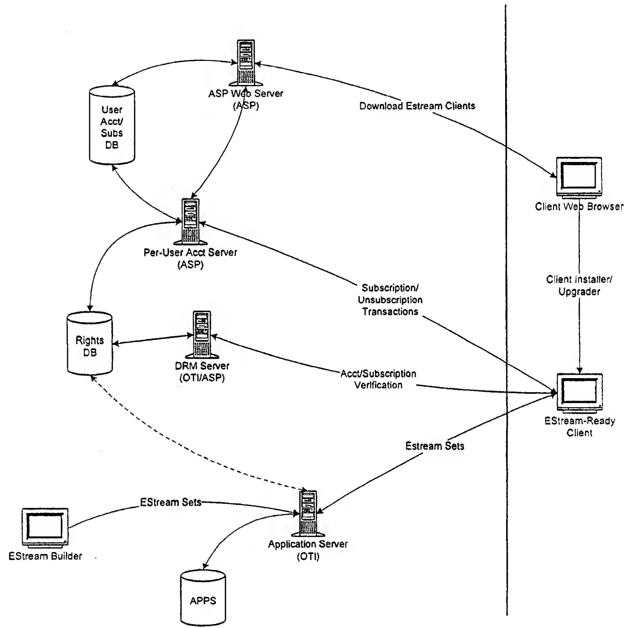
⇨ Supporting time-based charge for app-usage (e.g., rent by minutes of usage) complicates the design & may engender customer support/satisfaction issues. Do ASPs want/need this support? [Prefer to steer them away from this model.]

⇨ How should we handle minor upgrades/patches of apps (i.e., service packs)? One method to allow active use of previous versions plus availability of new versions without treating new versions as if they were entirely new applications would be file versioning.

EStream 1.0 Top Level Component Breakdown * Revision 0.1 * [redacted]
Client system components

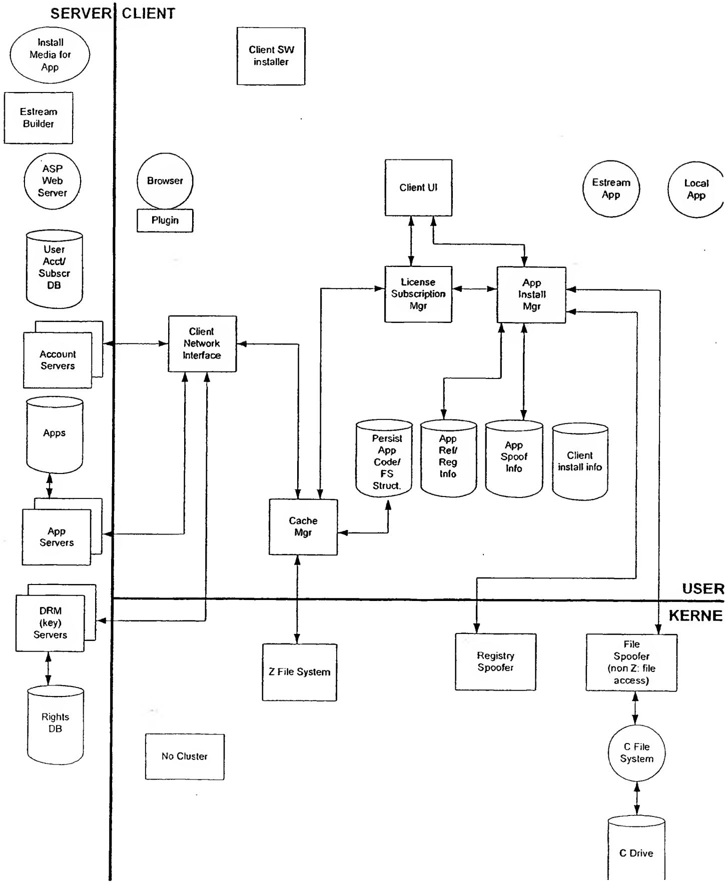
- ⇨ Z: File system manager [1]
    - o Handles all z: file system requests generated on client
    - o Makes requests to EStream cache manager
    - o Attempts to filter references that suggest software piracy activity
- ⇨ EStream client core
    - o Session manager [12]
        - Handles establishing/terminating ASP sessions
        - Negotiates for app license & security using user unique certificate
        - Invoked either by eStream client user interface or by cache mgr
    - o Cache manager [2]
        - Responds to Z: file system manager requests
        - Maintains client cache of app & file system data/metadata
        - Requests info as necessary from Estream client networking
        - Requests session/license for non-mounted apps from session mgr
        - Consumes/gathers profiling/feedback data
    - o File manager [3]
        - Provides interface to all eStream created/maintained client files
        - Gets requests from cache mgr, session mgr, file mgr/spoofer, registry mgr/spoofer, app install/deinstall, client install/deinstall
- ⇨ Estream client network interface [8]
    - o Handles requests from EStream cache manager
    - o Handles protocol interface to/from server
    - o Performs compression/decompression, encryption/decryption of packets
    - o Detects network problems & reports to session manager for renegotiation
- ⇨ EStream client user interface [5]
    - o Displays error/info messages from any part of eStream code to user
    - o Solicits/obtains info (e.g., login/password, app license) from user
- ⇨ EStream file system manager/spoofer [6]
    - o Filters all non-z: file sys requests, redirects non-z: file refs as appropriate
    - o Supports operation of eStreamed apps
    - o Avoids eStreamed apps interfering with non-eStreamed apps
- ⇨ Estream registry manager/spoofer [7]
    - o Filters all registry refs, handles registry contents for/about eStreamed apps
- ⇨ EStream application installer/deinstaller [14]
    - o Obtains app spoofing/registry/prefs info & initial cache/profile data
    - o Prepares system to be able to run app on user request
    - o Supports deinstallation of app
- ⇨ EStream client code installer/deinstaller [13]
    - o Installs all client Estream code components
    - o Supports deinstallation of all eStream components
- ⇨ NoCluster.sys [4]
    - o Disables page fault clustering in the kernel
- ⇨ Estream browser plugin
    - o Optional EStream component which fields unsolicited server messages

eStream Client-Server Diagram



v 0.1

eStream High-Level Design Diagram



eStream File System Straw Man Proposal

Version 0.5

[redacted]

Purpose

The purpose of this document is to present a concrete proposal for the functioning of the eStream file system. In many places, I make some sweeping generalizations about how things should work without describing the data structures and interfaces involved in implementing them. This document should eventually involve into a design specification.

Issues Not Covered

This document does not attempt to cover all issues present in designing the eStream 1.0 product. In particular, the overall authentication/licensing/security architecture is not covered in detail here. It is expected that the security functionality will be mostly orthogonal to the design of the basic file system functionality.

Background

There are a number of different networked file systems out there. Many of them share some requirements with eStream. For example, AFS performs client-side on-disk caching, while Coda handles serious server redundancy and disconnected operation. Personally, I believe that AFS and Coda are the file systems whose designs are most relevant to us. For those interested in further background reading, you might also want to look at papers covering NFS, CIFS, xFS, DFS, and Zebra.

Single File System Name Space

Many modern distributed file systems present the network file system as a single tree mounted at some location on the client system, regardless of which server hosts the data. (In fact, with AFS, every file on every server in the world can be accessed through a path starting with /afs on the client, assuming the client can reach that server and has sufficient privileges to do so.) Compared with systems like NFS and Windows sharing, where each share is mounted in a different location on the client, the single name space provides greater ease of use.

The eStream file system would present one universal logical file system. Regardless of which ASP provider supplies a particular volume, that volume will always be referenced via the same path on the eStream file system. That this is desirable or even feasible is predicated on the assumption that OTI is the only entity providing all eStream sets. Each volume must get a unique identifier and a unique location to be mounted in the file system hierarchy. If two different ASPs privide the same volume ID, then the contents of those volumes must be identical. This way, we don't have to tag things in the cache based on what ASP they came from, and the cache manager doesn't need to know anything about ASPs. If done correctly, only the client networking component and the LSM need to know about ASPs.

Volumes

A volume is a complete subtree of a file system. Volumes may contain files and directories. Volumes may not be mounted in other volumes. A volume is a logical grouping of files within the file system and is the unit of replication across servers. An application will reside in a single volume. Two applications will never share a volume.

Volumes are uniquely identified by a 32-bit volume identifier. Each volume additionally has an 8-bit version number. This version number is incremented each time any file within the volume changes. (See supporting upgrades, below). Note that the volume id is globally unique. If two ASPs provide volumes with the same volume number (and version), they have identical contents.

A volume may be replicated on any number of servers. Each SLM server contains a map describing the application servers that currently provide each volume. This global replication of this table is acceptable because volumes are added or moved infrequently.

Identifying Files

Files and directories are uniquely identified by the pair (volume id, file number). This tuple is called a file id. Volume id and file number are each 32-bit signed integers. Negative values for both volume id and file number are reserved for special purposes, leaving us with 2^31 possible volume IDs and 2^31 possible files per volume.

Finding an Application Server for a Volume

The SLM will tell the client which application servers currently provide each volume. It may be necessary for the client to periodically poll the SLM to get up-to-date information about the state of the application servers. The License and Subscription Manager on the client will keep track of the currently subscribed applications and the application servers for each of these applications.

Directories

Directories are specially formatted files that are used in a special way by the file system. They are identified by file ids, just like other files. From a client-server point of view, they are read by the client in the same way as other files. Directories contain arrays of entries with the following format:
(volume number, file number, flags, length, filename)

The volume number and file number are 32-bit signed integers. The flags are 32-bits of flags. The length is 16 bits and is the length of the filename in bytes. The filename is a non-NUL terminated Unicode string. The structure is padded with enough Unicode NUL characters to make the structure a multiple of 32 bits long. The next directory entry begins on the next 32-bit boundary.

The access token is not part of the directory, as a single access token is required to access all files in a particular volume.

The volume number is required so that the the client can construct a local directory for the root of the directory structure in the same format as other directories (see filename parsing below). It also helps to provide a sanity check.

Accessing Files

Assuming that a client has a file-id for a file that it wishes to access, the following client-server actions must be supported:

For stat-like information on the file, we need a GetFileMetadata() interface. The client would provide the file id it is interested in and the proper access token for this file. The server will either return the metadata for the file or an error condition (like access token expired or incorrect access token.) The metadata contains the standard Windows metadata information, including file length and file access times.

On a file open (CreateFile in Windows terminology), we need to verify that we have access to the requested file. This is probably best accomplished by calling GetFileMetadata and verifying that we can get the metadata. This way, we can fail file opens gracefully if we don't have an access token.

On reads (and writes, when we support them), the client will send the file id and the access token to the server along with an offset and a length for the read and write. The server will respond with the data. Note that the same mechanism will be used for reading both files and directories.

Pseudodirectories

For those parts of the eStream file system name space that do not belong to any volume (such as the root of the file system), the client must construct appropriate directories based on the currently installed applications. This is to support filename parsing starting at the root of the directory. For example, if the client has word installed with a root of /Worddir and it is volume number 3 and Photoshop installed with a root of /Photodir and is volume number 4, the client would construct a directory for the root of the entire file system containing
File name, Volume number, file number
"Worddir", 3, 0
"Photodir", 4, 0

(The file numbers are both zero here because 0 is the index of the root directory of each volume, and these are the mount points for each volume.)

When new applications are installed, the root of the file system would have to be updated to reflect the newly installed apps.

Filename Parsing

Filename parsing is handled one element at a time, starting at the root of the file system. Parsing one path name element involves reading the parent directory's contents (from the

cache or the app server), searching it for the file matching the next path element's name, and getting the appropriate file id so it can do further lookup.

Volume Versioning... Without File Versioning
We can provide volume versioning and incremental volume updates without versioning each file in the file system. When a new volume is to be provided, we can append any new or changed files as new files in the volume, with new volume IDs that weren't already present. If a directory's contents have changed, then a new version of this directory will be built, with a new file number. This process will proceed from the leaves all the way to the root of the file system, eventually resulting in a new root. The old versions of things would still be available for old clients to access, but clients wishing to access the new version will simply start at the new root, and would thereby get to a consistent picture of the volume. Any file or directory that has not changed from the old version to the new one need not be replicated, and will be referenced by its old file number. (I.e. newly reconstructed directories will contain the old file number for any files that haven't changed.)

If we reserve the first 256 file ids for the root directory, then the version number can be the same as the file number for the root directory.

Note that if we decide that the complexity of this approach is too high, this does not preclude always creating a new volume from scratch for each update.

Constructing File IDs
It is the job of the builder to produce the volume file to file id mapping and to construct all of the directories. Because directories are files identified by file id, this process must begin at the leaves of the volume and proceed to the root.

Note that constructing a new changed volume will consist of finding the diffs between the two volumes and producing some new directories. Changed or newly added files will get new file numbers, leaving the old ones around. Note that any directory that has had any descendents changed must be reconstructed with the new file numbers, and the new directory will get a new file number. This process will proceed to the root of the volume, which will receive a new file number.

Server Failover
All app servers for a particular volume must share the same mapping of file ids to file, so server failover is trivial. There might be a performance impact if the new app server doesn't have the requrested file in memory.

Writing Files into the Application Install Directories
Two approaches have been discussed for the problem of applications that want to write files to their install directories. First, this can be handled wholly inside of the eStream file system. The cache manager could allow writes to files handled by the eFS, but these writes would not be written back to the server. Instead, they would simply be written to

the eFS cache and marked non-purgeable. This approach's primary advantage is that it does not rely on a file spoofer.

The other approach is to use the file spoofer to spoof some accesses to the z: drive. Any open for read/write access would cause the existing file (if any) to be copied to a location on the c: drive, and the file spoofer would then redirect the open to the newly created file. The file spoofer would have to keep track of any file created via this copy-on-write mechanism and redirect all future accesses to the copy. There are some issues to this approach. For example, it is extremely wasteful when files on the z: drive are opened for read/write access but are never actually written. However, it does help reduce the complexity of the eFS cache, and is trivial to implement if we have to do c: to z: file spoofing anyway.

In either case, to support the creation of new files in an application's install directory, it must be possible to modify the contents of directories in the cache.

If we don't use the file spoofing approach, there is the issue of how we support written files when we move to a newer version of a volume. It would probably be necessary to walk the cache and make sure that each written file gets placed in the appropriate place in the new volume version. This is likely to be non-trivial, because we need to have full information about the location of each modified file in the file system tree, and would need to download enough of the new volume directory structure to place these modified files there.

64-Bit File Access?

One question we should answer is whether we will support file sizes greater than 2 GB on the eStream file system. I'm inclined to say that such support isn't a requirement for the 1.0 product, but I also think that the implementation and verification complexity of 64-bit file access on the file system is low enough that we might want to consider building it in anyway.

Simplifications

We could preclude the possibility of an application consisting of more than one volume.

Future Possibilities

Epicon seems to make a big selling point of their technology involving "self-healing" of damaged application files. Such support could be provided by computing checksums on files in the cache. Whether or not we want to support this is an open question. My feeling is that it's something we should leave out of 1.0.

Outstanding Issues

Cache organization has not been addressed.

Finding and downloading the app install block has not been addressed.

Security in a multiuser system has not been addressed.

eStream File Spoofer Low Level Design

Curt Wohlgemuth
Version 2.0
[redacted]

Functionality

The eStream file spoofer is a kernel-mode driver responsible for redirecting file accesses from local file systems to the eStream file system driver. It is implemented as a file system filter driver that traps all IRP requests to the file system device handling drives that must be spoofed, and redirects these requests to the EFSD.

Data type definitions

The file spoofer will understand entries in the "file spoof database" as they have been identified by the eStream builder and installed by the app install manager, but these are not defined by this component.

Entries in this spoof database will have the following entries:

- original (fully qualified) path name of file: this resides somewhere on a local disk of the client machine
- new (fully qualified) path name of the file to spoof to: this resides on the eStream file system drive

The spoof database will reside in the registry, so it can be persistent across reboots, and so the file spoofer need not open a file to load it. As applications are installed on a client machine, the Application Install Manager will load new spoof entries into the registry, and inform the file spoofer that it must reload this database. Similarly, when an app is uninstalled, the AIM will remove spoof entries from the registry, and inform the file spoofer.

This proposes that the each spoof entry is a separate name/value entry under a single key in the registry:

- Name: the original filename
- Value: the new filename

Interface definitions

The eStream file spoofer is called by two components:

1. The eStream client start service, which will start and stop the file spoofer

2. The AIM, which will inform the file spoofer to reload the spoof database from the registry

The interfaces called by both of these user-mode components will be in the form of DeviceIoControl() calls. The following IOCTL codes will be defined for use by callers of the file spoofer:

```
IOCTL_FS_START_SPOOFING
IOCTL_FS_STOP_SPOOFING
IOCTL_FS_RELOAD_SPOOF_DB
```

Starting spoofing

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that starting spoofing is currently identical to reloading the spoof database.

This is called by the eStream client start service on startup.

Stopping spoofing

The input and output buffers for this IOCTL are ignored and should be NULL. This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to clear its memory image of the spoof database.

This is called by the eStream client start service on shutdown.

Reload spoof database

The input buffer for this IOCTL should supply the name of the registry key containing the spoof entries as values. The output buffer for this IOCTL is ignored and should be NULL.

This will return either STATUS_SUCCESS, or an error return status if something goes wrong. It causes the file spoofer to read the spoof registry entries, and load up each entry into memory.

Note that this is currently identical to starting the spoof database.

This is called by the AIM when a new eStream app is installed.

Component design

The file spoofer will have these major tasks:

- Track the following data:
  - all current valid file spoof entries
  - spoof entries by filtered file system
- Filter native file system drivers for local drives, intercept all IRP_MJ_CREATE and FAST_IO_QUERY_OPEN requests, and for spoofed files, change the file-name of the FileObject associated with these requests.

Data structures

The file spoofer needs to be able to quickly look up a filename in the in-memory spoof database. The current design will use a hash table, whose size and hash function will be tuned as we get experience with real applications.

Adding or deleting entries from the hash table will by synchronized using a global resource.

Algorithms

Here are basic algorithms for these steps.

Load spoof database

This reads all the name/value pairs under the registry key which holds the spoof entries, loads them into a temporary hash table, then points the real hash table to this one.

```
traverse the registry until the input registry key is opened, using ZwOpenKey()
if not found
    return error
if no name/value pairs exist in this key
    return "no data"

for each name/value pair found with ZwEnumerateValueKey()
    build a hash node for this and insert it into temp hash table
    if the drive letter for the old filename entry is one we're not currently filtering
        put this drive letter on new drive list

acquire the hash table resource exclusively
point the global hash table head pointer to the temp hash table
release the resource

for each drive letter on the new drive list
```

```
        look up FS device for this drive
        if we really aren't attached to it
            attach self to this FS device as filter driver

    free the old hash table
    free the drive list
    return success
```

Stop spoofing

```
    acquire hash table resource exclusively
    free the global hash table
    detach self from all filtered FS devices
    release hash table resource
```

Trap Create and QueryOpen requests

```
    acquire the hash table resource shared
    if hash table head pointer is non-NULL
        look up input filename in hash table
    release hash table resource
    if filename not found in hash table
        send I/O request to original target FS driver
    else
        free memory of existing file name in input FileObject
        allocate memory for new, spoofed filename, copy into this memory
        send I/O request to eStream file system driver
```

Testing design

Unit testing plans

The file spoofer will be tested as a standalone component, apart from the rest of the eStream client.. A driver test program will be written to test all functionality and corner cases. This includes filtering all FSDs active for a client system, and multiple drives handled by a single FSD.

Stress testing plans

The file spoofer should be able to work, with little or no performance cost to the system as a whole, even when the attached FSDs are under heavy load. Some stress testing will be done in this fashion.

Coverage testing plans

If we come up with a method for measuring coverage for kernel-mode components, we'll do so for the file spoofer as well.

Cross-component testing plans

Not clear if anything need be done here outside of the standard execution of the eStream client.

Upgrading/Supportability/Deployment design

I don't see any upgrade/compatibility issues for the file spoofer. For supportability, there will be a good debugging strategy and sufficient error message return codes for the caller.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. We will need to experiment with the hash table to tune it for fast lookup. It's possible that we may need to replace the hash table with a faster lookup algorithm.

SCENARIO: Install a subscribed application

LEGEND: use of specialized fonts in material below:
Bold indicates block or entire scenario
Bold italic indicates argument or return interface between blocks or scenarios
Italic designates meta-info about scenario

BACKGROUND: ways of invoking this scenario

- ➪ User subscribes to a new application or requests upgrade of existing app.
- ➪ User starts using a new client.
- ➪ Client learns that a new application is available.

DISCUSSION: Nature of the **AppInstallManager**
The **AppInstallManager** block may not be a generic module installed as part of the client installation of eStream; instead, it may be a unique executable associated with each particular application. Either way, it is expected that there is certain basic functionality associated with this code; that functionality is described below.

DISCUSSION: Need for checking license at install time
The scenario below describes that the application's license is checked at install time. This may need to be discussed further. Two reasons for license check at install time are:

(1) may be required by the software vendor's licensing model, and
(2) allows eStream client to record which ASP subscription to use when app invoked.

AppInstallManager: invoked w/***DRM server name & application serial number***

- ➪ *ACTION*: Establish connection, get license.
  - o Perform **CheckLicense**, which involves asking **ClientNetworkInterface** to send ***check-license*** message including ***application serial number*** to ***DRM server name*** (securely) & getting back ***response***.
    - If response = ***License ok***, then ***App Server name & session id*** returned, along with ***application nickname*** and ***upgrade flag***. If upgrade flag ***set***, ask **ClientUI** to display ***message*** advising user about upgrade & to solicit ***response*** concerning whether user wants to continue with current version. If response ***affirmative*** or if upgrade flag ***not set***, continue with next *ACTION*. Else, return ***status*** to **AppInstallManager** caller.
    - If response = ***License not ok***, then ***reason not ok*** returned. Some possible reasons license may not be ok include:
      - app license ***expired***. Ask **ClientUI** to display ***message*** advising user about license status & suggesting that user go to the ASP web server & renew license. Return ***status*** to **AppInstallManager** caller.
      - app not accessible (DRM server did not respond, DRM server indicated that application no longer supported, etc). Ask **ClientUI** to display ***message*** advising user that app not

available & suggesting that user go to the ASP web server for more info. Return *status* to **AppInstallManager** caller.

- ⇨ *ACTION*: Get application installation information.
  - o Perform **GetAppInstallBlock**, which involves asking **ClientNetworkInterface** to send ***get-install-info*** message including ***session id*** to ***App Server name*** (securely) & getting back ***response***.
    - If response = ***success***, then pointer to allocated ***AppInstallBlock*** data is also provided. Control continues with next *ACTION*.
    - If response = ***failure***, then ***status*** returned to **AppInstallManager** caller.
  - o Please note that if the **AppInstallManager** is an application-specific program, it may not request the entire ***AppInstallBlock*** contents at once.

- ⇨ *ACTION*: Check if application already installed.
  - o Perform **CheckIfAppAlreadyInstalled**, which involves using relevant info supplied as part of ***AppInstallBlock*** to check for the presence of certain registry entries and/or files. This is expected to detect both previous estream & non-estream installations in effect. If app ***not already installed***, continue with next *ACTION*. If app ***already installed***, ask **ClientUI** to display ***message*** advising user about this & to solicit ***response*** indicating whether to continue with current installation. If response ***negative***, then return ***status*** to **AppInstallManager** caller. If response ***affirmative***, continue with next *ACTION*.

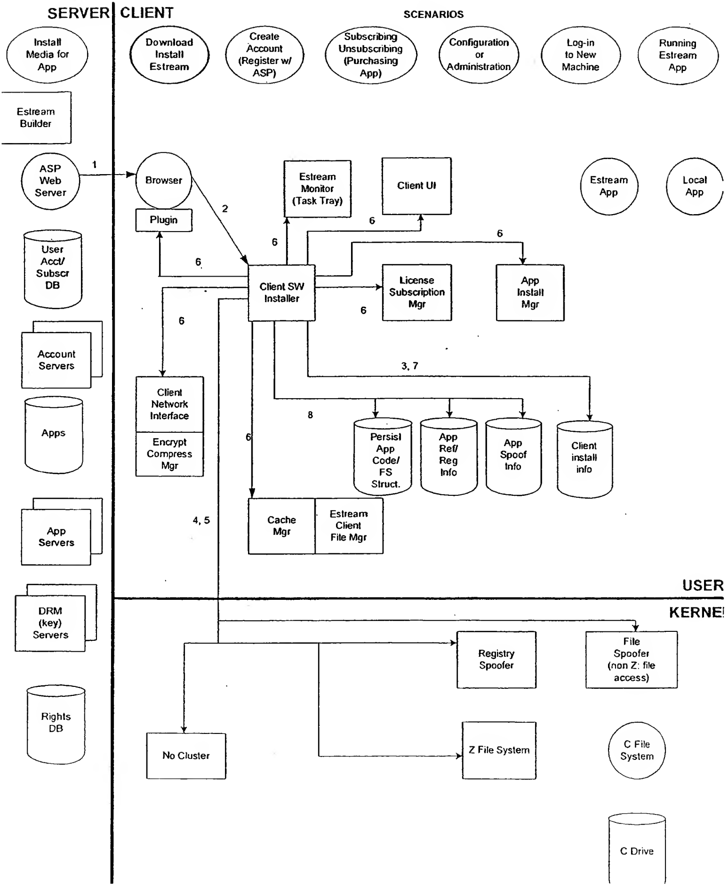
- ⇨ *ACTION*: Parse ***AppInstallBlock*** info, Set up client to make app ready to run
  - o *ACTION*: Handle non-Z: file association information in ***AppInstallBlock***.
    - Pass ***ptr*** to non-Z: file association data in ***AppInstallBlock*** to **InitializeFileSpooferData**, which checks client's system against relevant files (creating any non-Z: files that need to exist initially) and which passes that ***ptr*** along with ***filespoofer-data*** designation and ***application nickname*** to **eStreamClientFileMgr** for storage in appropriate area on client.
  - o *ACTION*: Handle registry information in ***AppInstallBlock***.
    - Pass ***ptr*** to registry data in ***AppInstallBlock*** to **InitializeRegistrySpooferData**, which modifies client's registry as appropriate and which passes that ***ptr*** along with ***registry-data*** designation and ***application nickname*** to **eStreamClientFileMgr** for storage in appropriate area on client.
  - o *ACTION*: Handle application data in ***AppInstallBlock***.
    - Pass ***cache-data*** designation, ***application nickname***, & ***ptr*** to initial application cache data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.
    - Pass ***profile-data*** designation, ***application nickname***, & ***ptr*** to initial profile data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.

        - For any application files to be preinstalled, pass ***file-data*** designation, ***application nickname, application filename,*** & ***ptr*** to file data in ***AppInstallBlock*** to **eStreamClientFileMgr** for storage in appropriate area on client.
    - *ACTION*: Perform other application-specific activities as desired.
        - If the **AppInstallManager** is generic code, then there is an interface to download an extra executable to do additional activities. If the **AppInstallManager** is code specific to the application, optional extra activities are included in that executable.

⇨ ACTION: Record app installation on client system
- Have **eStreamClientFileMgr** record ***application nickname, DRM server name,*** & ***application serial number*** in database of apps installed on client.

⇨ ACTION: Return status to **AppInstallManager** caller.

Scenario 1: Initial install



eStream 1.0 License Subscription Manager (LSM)

Omnishift Technologies, Inc.
Company Confidential

Functionality

This component is a COM Server executable.

The LSM manages the users subscriptions to the different ASP accounts. It is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running and is always active when the client component is running. Users on a given machine establish a connection with the ASP account servers from which they have subscribed applications. Users can add and delete the applications that are subscribed from the ASP accounts. The LSM makes the appropriate calls to the account servers to perform those actions. It gets serial numbers for the applications that are being subscribed and deletes them for the applications being un-subscribed (which are all part of the ASP ID Block). When the users start running any of the subscribed eStream applications, the eStream file system first queries the LSM before servicing any requests. The LSM in turn gets the appropriate access tokens from DRM servers along with the identities of application servers that can be used to run the applications. It uses the client identification (serial number) obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache. The eStream Cache Manager informs the LSM when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Data type definitions

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.

Field Name	Type
ASP ID	GUID
ASP NAME	BSTR
ASP URL	BSTR
ASP IP	DWORD

2. The ASP Subscription Blocks are created when the user establishes an account with the ASP service. These blocks enables secure logon to the ASP Server.

Field Name	Type
USER ID	GUID
USER NAME	BSTR
USER HASH PASSWORD	BSTR

3. Application Subscription Blocks are created for every application that is subscribed. These blocks are created when the application subscription is started and are updated when the application is run.

Field Name	Type
APPLICATION ID	GUID
APPLICATION NAME	BSTR
RATE	CURRENCY
PERIOD	INTEGER

4. The access tokens and the identities of the applications servers that are obtained from the DRM servers when the user tries to run the applications.

Field Name	Type
TOKEN ID	GUID
APPLICATION ID	GUID
EXPIRATION	DATE
TOKEN SIZE	DWORD
TOKEN DATA	BYTE *

Interface definitions

Subscription Management

Subscription management is the main interface between the Client UI control panel and the License Subscription manager. Tables containing lists of Application Service Providers and Subscribed Applications are managed using the Subscription manager interface.

ILicenseSubscritpionManager::IDispatch

The LSM exposes the following set of APIs to the client UI.

BOOL SubscribeApp(GUID & ASPId, GUID & AppID, LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

HRESULT UnsubscribeApp(ASPId, AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

HRESULT GetNextAppID(GUID & AppID)

This routine will return a pointer to a list of subscribed applications on the client machine.
The LSM exposes the following set of APIs to the eStream file system.

HRESULT CheckAccess(Path, &Root)

The LSM establishes a co-relation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the DRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "root".

BeginApp(AppID)

To indicate the start of an application.

EndApp(AppID)

To indicate the end of the application.

The LSM makes the following API calls.

1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM also sends messages to the account server for subscribing and unsubscribing applications and to the DRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from the account server when the user first establishes a connection with it.

ISubscriptionManager

ISubscription Methods	Description
SubscribeApp	Subscribes an eStream Application.
UnsubscribeApp	Un-Subscribe an eStream Application
CheckAccess	Check to see if an eStream application is subscribed
BeginApp	Begin an eStream Application
GetNextApp	Get the next application the ASP Supplies
GetNextSubscribedApp	Get the next Subscribed Application
GetLicenseInfo	Get the license info for the application.

HRESULT ISubscriptionManager:: SubscribeApp

```
HRESULT SubscribeApp(
    GUID AppID,
    GUID AspID,
    BSTR *licenseInfo,
   );
```

Parameters

AppID
[in] Identifier for application to be subscribed.

AspID
[in] Identifier for the ASP service that the application is going to be subscribed to.

licenseInfo
[out] License info block for the subscribed application.

Return Values

Returns NOERROR if successful, or an OLE-defined error value otherwise.

Remarks

This function will return an error is a user attempts to subscribe an application that is already subscribed.

HRESULT ISubscriptionManager:: UnSubscribeApp

```
HRESULT SubscribeApp(
    GUID AppID
        );
```

Parameters

AppID
[in] Identifier for application to be un-subscribed.

Return Values

Returns NOERROR if successful, or an OLE-defined error value otherwise.

Remarks

This function will return an error is a user attempts to un-subscribe an application that is not subscribed.

BOOL ISubscriptionManager::CheckAccess

```
HRESULT CheckAccess(
    GUID AppID
        );
```

Parameters

AppID
[in] Identifier for application to be checked for access

Return Values

Returns TRUE if application can be access, FALSE otherwise.

BOOL ISubscriptionManager:: BeginApp

```
HRESULT BeginApp(
    GUID AppID
       );
```

Parameters

AppID
    [in] Identifier for application to be started

Return Values

Returns S_OK if application can be started, E_NOACCESS if the application is not subscribed.

BOOL ISubscriptionManager:: GetNextApp

```
HRESULT GetNextApp(
    GUID AspID
    GUID AppID
       );
```

Parameters

AspID [in] ASP Account ID to check for application

AppID
    [out] Identifier for application to be queried for ID this will be null when the list of applications runs out.

Return Values

Returns S_OK if application can be started, E_NOACCESS if the application is not subscribed.

Token Management – Overview

Token management is the other major function that the eStream License Manager provides. These tokens are requested from the eStream Server every time an eStream application is started and release when an eStream application is terminated. Access tokens are issued for a finite period and renewed on a periodically depending on their expiration timestamp.

Token Management – Cache Manager Interface

Token management is the service that the License Manger provides to the cache manager. A table will be used store access tokens that the Cache manager uses. The ITokenManger interface provides access to tokens.

ITokenManager::IDispatch

The LSM exposes the following set of APIs to the Cache Manage.

HRESULT ITokenManager::GetToken

```
HRESULT GetToken(
     GUID AppID,
     GUID AspID,
     DWORD expires,
     GUID & TokenID,
     DWORD Tokensize,
     BYTE * Tokendata
    );
```

Parameters

AppID
    [in] Identifier for application to be subscribed
AspID
    [in] Identifier for the ASP service that the application is going to be subscribed to
Expires
    [out] Time interval for which the token is valid
TokenID
    [out] Token ID
TokenSize
    [out] Size of the token data
Tokendata
    [out] token data

Return Values

Returns S_OK if successful, an expired token returns an error.

Remarks

Acquire a token for the License manager.

HRESULT ITokenManager::ReleaseToken

```
HRESULT ReleaseToken(
    GUID TokenID,
   );
```

Parameters

TokenID
[in] ID for token to be released

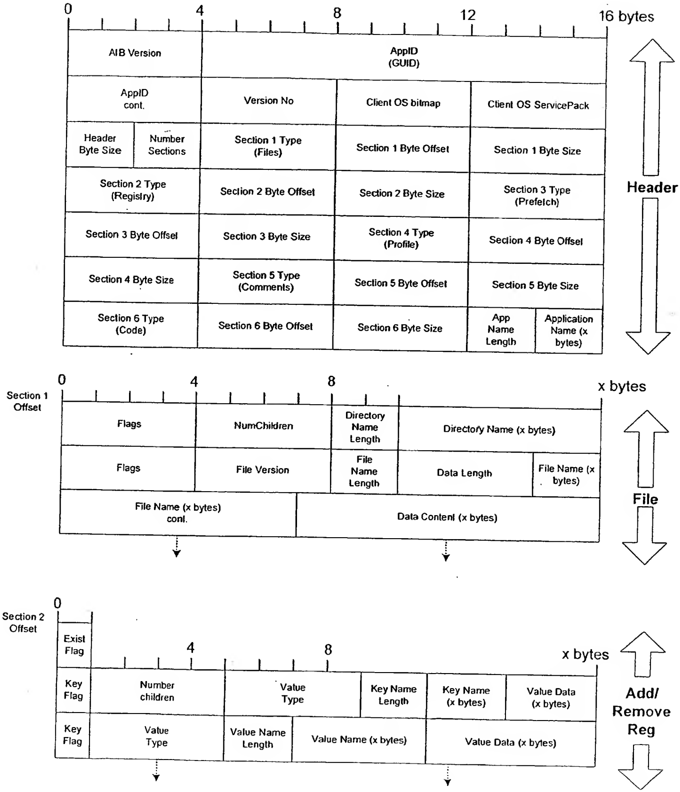
Return Values

Returns S_OK if successful, an expired token returns an error.

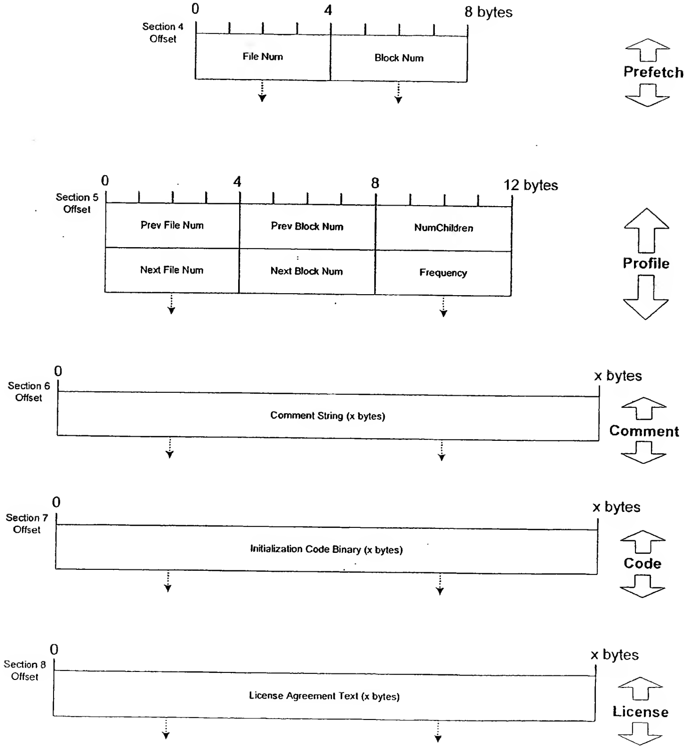
Remarks

Release a token for the License manager.

Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



eStream AppInstallBlock Low Level Design

Sanjay Pujare and David Lin
Version 0.2
[redacted]

Functionality

The AppInstallBlock is a block of code and data associated with a particular application. This AppInstallBlock contains the information needed to by the eStream client to 'initialize' the client machine before the eStream application is used for the first time. It also contains optional profiling data for increasing the runtime performance of that eStream application.

The AppInstallBlock is created offline by the eStream Builder program. First of all, the Builder monitors the installation process of a local version of the application installation program and records changes to the system. This includes any environment variables added or removed from the system, and any files added or modified in the system directories. Files added to the application specific directory is not recorded in the AppInstall-Block to reduce the amount of time needed to send the AppInstallBlock to the eStream client. Secondly, the Builder profiles the application to obtain the list of critical pages needed to run the application initially and an initial page reference sequence of the pages accessed during a sample run of the application. The AppInstallBlock contains an optional application-specific initialization code. This code is needed when the default initialization procedure is insufficient to setup the local machine environment for that particular application.

The AppInstallBlock and the runtime data are packaged into the eStream Set by the Builder and then uploaded to the application server. After the eStream client subscribed to an application and before the application is run for the first time, the AppInstallBlock is send by the server to the client. The eStream client invokes the default initialization procedure and the optional application-specific initialization code. Together, the default and the application-specific initialization procedure process the data in the AppInstall-Block to make the machine ready for eStreaming that particular application.

Data type definitions

The AppInstallBlock is divided into the following sections: header section, variable section, file section, profile section, prefetch section, comment section, and code section. The header section contains general information about the AppInstallBlock. The information includes the total byte size and an index table containing size and offset into other sections. In Windows version, the variable section consists of two registry tree structures to specify the registry entries added or removed from the OS environment. The file section is a tree structure consisting of the files copied to C drive during the application installation. The profile section contains the initial set of block reference sequences during

Builder profiling of the application. The prefetch section consists of a subset of profiled blocks used by the Builder as a hint to the eStream client to prefetch initially. The comment section is used to inform the eStream client user of any relevant information about the application installation. Finally, the code section contains an optional program tailored for any application-specific installation not covered by the default eStream application installation procedure. In Windows version, the code section contains a Windows DLL.

Here is a detailed description of each fields of the AppInstallBlock.

Note: Little endian format is used for all the fields spanning more than 1 byte. Also, BlockNumber specifies blocks of 4K byte size.

1. Header Section:

The header section contains the basic information about that AppInstallBlock. This includes the versioning information, application identification,

Core Header Structure:

- **AibVersion [4 bytes]**: Magic number or appInstallBlock version number (which identifies the version of the appInstallBlock structure rather than the contents).
- **AppId [16 bytes]**: this is an application identifier unique for each application. On Windows, this identifier is the GUID generated from the 'guidgen' program. AppId for Word on Win98 will be different from Word on WinNT if it turns out that Word binaries are different between NT and 98.
- **VersionNo [4 bytes]**: Version number. This allows us to inform the client that the appInstallBlock has changed for a particular appId. This is useful for changes to the AppInstallBlock due to minor patch upgrades in the application.
- **ClientOSBitMap [4 bytes]**: Client OS supported bitmap or ID: for Win2K, Win98, WinNT and other future OSs we might support (it should be possible to say that this appInstallBlock is for more than one OS).
- **ClientOSServicePack [4 bytes]**: We might want to store the service pack level of the OS for which this appInstallBlock has been created. Note that when this field is set we cannot use multiple OS bits in the above field ClientOSBitMap.
- **Flags [4 bytes]**: Flags pertaining to AppInstallBlock
  - **Bit 0:** Reboot – If set, the eStream client needs to reboot the machine after installing the AppInstallBlock on the client machine.
  - **Bit 1:** Unicode – If set, the string characters are 2 bytes wide instead of 1 byte.
- **HeaderSize [2 bytes]**: Total size in bytes of the header section.
- **Reserved [32 bytes]**: Reserved spaces for future.

- **NumberOfSections [1 byte]**: Number of sections in the index table. This determines the number of entries in the index table structure described below:

Index Table Structure: (variable number of entries)

- **SectionType [1 bytes]**: The type of data describe in section. 0=file section, 1=variable section, 2=prefetch section, 3=profile section, 4=comment section, 5=code section.
- **SectionOffset [4 bytes]**: The offset from the beginning of the file indicates the beginning of section.
- **SectionSize [4 bytes]**: The size in bytes of section.

Variable Structure:

- **ApplicationNameLength [4 bytes]**: Byte size of the application name
- **ApplicationName [X bytes]**: Non-null terminating name of the application

2. File Section:

The file section contains a subset of the list of files needed by the application to run properly. This section does not enumerate files located in the standard application program directory. It consists of information about files copied into 'unusual' directory during the installation of an application. If the file content is small, the file is copied to the client machine. Otherwise, the file is relocated to the standard program directory suitable for streaming. The file section data is list of trees stored in a contiguous sequence of address space according to the pre-order traversal of the trees. A node in the tree can correspond to one or more levels of directory. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal Windows pathname including the drive letter. Each entry of the node in the tree consists of the following structure:

Directory Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 0 is set if this entry is a directory
- **NumberOfChildren [2 bytes]**: Number of nodes in this directory
- **DirectoryNameLength [4 bytes]**: Length of the directory name
- **DirectoryName [X bytes]**: Non-null terminating directory name

Leaf Structure: (variable number of entries)

- **Flags [4 byte]**: Bit 1 is set to 1 if this entry is a spoof or copied file name
- **FileVersion [4? bytes]**: Version of the file GetFileVersionInfo() if the file is win32 file image. Need variable file version size returned by GetFileVersionInfoSize(). Otherwise use GetFileTime() to retrieve the file creation time.
- **FileNameLength [4 bytes]**: Byte size of the file name

- **DataLength [4 bytes]**: Byte size of the data. If spoof file, then data is the string of the spoof directory. If copied file, then data is the content of the file
- **FileName [X bytes]**: Non-null terminating file name
- **Data [X bytes]**: Either the spoof file name or the content of the copied file

3. Add Variable and Remove Variable Sections:

The add and remove variable sections contain the system variable changes needed to run the application. In Windows system, each section consists of several number of registry subtrees. Each tree is stored in a contiguous sequence of address space according to the pre-order traversal of the tree. A node in the tree can correspond to one or more levels of directory in the registry. A parent-child node pair is combined into a single node if the parent node has only a single child. Parsing the tree from the root of the tree to a leaf node results in a fully legal key name. The order of the trees is shown here.

a. Registry Subsection:

1. "KHCR": HKEY_CLASSES_ROOT
2. "HKCU": HKEY_CURRENT_USER
3. "HKLM": HKEY_LOCAL_MACHINE
4. "HKU": HKEY_USERS
5. "HKCC": HKEY_CURRENT_CONFIG

Tree Structure: (5 entries)

- **ExistFlag [1 byte]**: Set to 1 if this tree exist, 0 otherwise.
- **Key or Value Structure entries [X bytes]**: Serialization of the tree into variable number key or value structures described below.

Key Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **NumberOfSubchild [4 bytes]**: Number of subkeys and values in this key directory
- **KeyNameLength [4 bytes]**: Byte size of the key name
- **KeyName [X bytes]**: Non-null terminating key name

Value Structure: (variable number of entries)

- **KeyFlag [1 byte]**: Set to 1 if this entry is a key or 0 if it's a value structure
- **ValueType [4 byte]**: Type of values from the Win32 API function RegQueryValueEx(): REG_SZ, REG_BINARY, REG_DWORD, REG_LINK, REG_NONE, etc...
- **ValueNameLength [4 bytes]**: Byte size of the value name
- **ValueDataLength [4 bytes]**: Byte size of the value data

- **ValueName [X bytes]**: Non-null terminating value name
- **ValueData [X bytes]**: Value of the Data

In addition to registry changes, an installation in Windows system may involve changes to the ini files. The following structure is used to communicate the ini file changes needed to be done on the eStream client machine. The ini entries are appended to the end of the variable section after the 5 registry trees are enumerated.

b. INI Subsection:

- **NumFiles [4 bytes]**: Number of INI files modified.

File Structure: (variable number of entries)

- **FileNameLength [4 bytes]**: Byte length of the file name
- **FileName [X bytes]**: Name of the INI file
- **NumSection [4 bytes]**: Number of sections with the changes

Section Structure: (variable number of entries)

- **SectionNameLength [4 bytes]**: Byte length of the section name
- **SectionName [X bytes]**: Section name of an INI file
- **NumValues [4 bytes]**: Number of values in this section

Value Structure: (variable number of entries)

- **ValueLength [4 bytes]**: Byte length of the value data
- **ValueData [X bytes]**: Content of the value data

4. Prefetch Section:

The prefetch section contains a list of file blocks. The Builder profiler determines the set of file blocks critical for the initial run of the application. This data includes the code to start and terminate the application. It includes the file blocks containing for frequently used commands. For example, opening and saving of documents are frequently used commands and should be prefetched if possible. Another type of blocks to include in the prefetch section is the blocks associated with frequently accessed directories and file metadata in this directory. The format of the data is described below:

- **FileNumber [4 bytes]**: File Number of the file containing the block to prefetch
- **BlockNumber [4 bytes]**: Block Number of the file block to prefetch

5. Profile Section: (not used in eStream 1.0)

The profile section consists of a reference sequence of file blocks accessed by the application at runtime. Conceptually, the profile data is a two dimensional matrix. Each entry [*row, column*] of the matrix is the frequency a block *row* is followed by a block *column*. In any realistic applications of fair size, this matrix is very large and sparse. Proper data structure must be selected to store this sparse matrix efficiently in required storage space and minimize the overhead in accessing this data structure access.

The section is constructed from two basic structures: row and column structures. Each row structure is followed by N column structures specified in the NumberColumns field. Note that this is an optional section. But with appropriate profile data, the eStream client prefetcher performance can be increased.

Row Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the row block
- **BlockNumber [4 bytes]**: Block Number of the row block
- **NumberColumns [4 bytes]**: number of blocks that follows this block. This field determines the number of column structures following this field.

Column Structure: (variable number of entries)

- **FileNumber [4 bytes]**: File Number of the column block
- **BlockNumber [4 bytes]**: Block Number of the column block
- **Frequency [4 bytes]**: frequency the row block is followed by column block

6. Comment Section:

The comment section is used by the Builder to describe this AppInstallBlock in more detail.

- **Comment [X bytes]**: Null terminating comment string

7. Code Section:

The code section consists of the application-specific initialization code needed to run on the eStream client to setup the client machine for this particular application. This section may be empty if the default initialization procedure in the eStream client is able to setup the client machine without requiring any application-specific instructions. On the Windows system, the code is a DLL file containing two exported function calls: *Install*(), *Uninstall*(). The eStream client loads the DLL and invokes the appropriate function calls.

- **Code [X bytes]**: Binary file containing the application-specific initialization code. On Windows, this is just a DLL file.

8. LicenseAgreement Section:

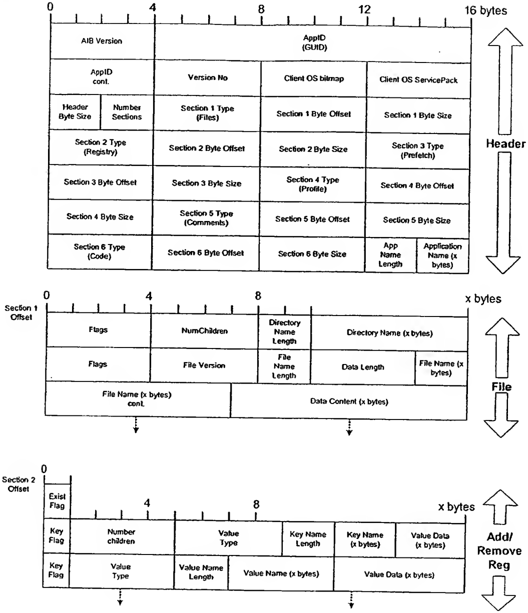
The Builder creates the license agreement section. The eStream client displays the license agreement text to the end-user before the application is started for the first time. The end-user must agree to all licensing agreement set by the software vendor in order to use the application.

- **LicenseAgreement [X bytes]**: Null terminating license agreement string

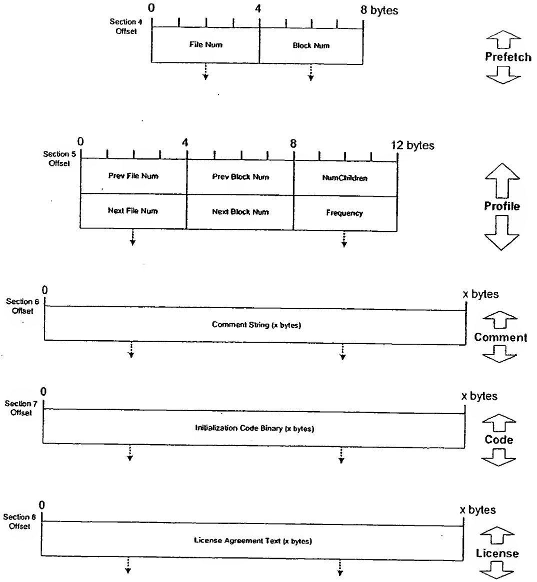
Open Issues

- What is the size of the AppInstallBlock for a typical application like Office?
- How large should the prefetch sections be for optimal run of an application? At minimum, it should contain at least start/termination code.
- How should the AppInstallBlock handle application license agreement text string? Add a new section or use comment section. Does the dialog need to have exactly the same interface as the license agreement dialog on the local installation?
- Currently, file section stores complete pathname including the drive letter. The installation may place files according to some variables like %SystemRoot% or %UserProfile%. How does the Builder detect this so it can propagate this information to the client?

Format of AppInstallBlock (part 1 of 2)



Format of AppInstallBlock (part 2 of 2)



The eStream Builder

The eStream Builder is a software program. It is used to convert locally installable applications into a data set suitable for streaming over the network. The streaming-enabled data set is called the eStream Set. This document describes the procedure used to convert locally installable applications into the eStream Set.

The application conversion procedure into the eStream Set consists of the several steps. In the first phase, the Builder program monitors the installation process of a local installation of the desired application for conversion. The Builder monitors any changes to the system and records those changes in an intermediate data structure. After the application is installed locally, the Builder enters the second phase of the conversion. In the second step, the Builder program invokes the installed application executable and obtains sequences of frequently accessed file blocks of this application. Both the Builder program and the eStream client software use the sequence data to optimize the performance of the streaming process. Once the sequencing information is obtained, the Builder enters the final phase of the conversion. In this step, the Builder gathers all data obtained from the first two phase and processes the data into the eStream Set.

In the next sections, detailed descriptions of the three phases of the Builder conversion process are described. The three phases consists of installation monitoring, application profiling, and finally eStream packaging. In most cases, the conversion process is general and applicable to all type of system. In places where the conversion is OS dependent, the discussion is focused on Microsoft Windows environment. Issues on conversion procedure for other OS environment are described in later sections.

Installation Monitoring

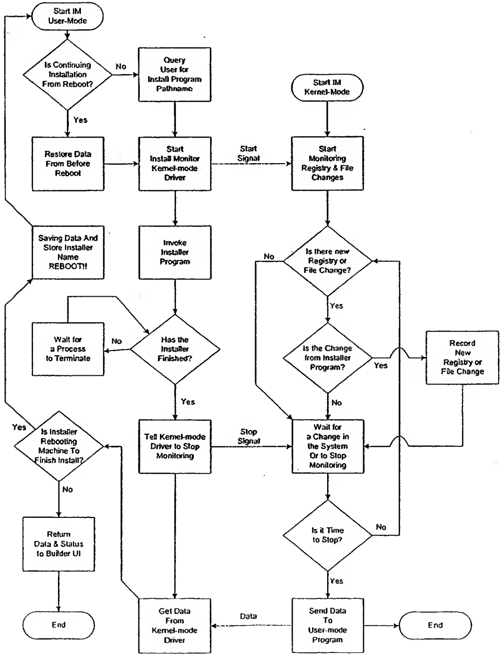
In the first phase of the conversion process, the Builder Installation Monitor (IM) component invokes the application installation program that installs the application locally. The IM observes all changes to the local computer during the installation. The changes may involve one or more of the following: changes to system or environment variables; and modifications, addition, or deletion of one or more files. The IM records all changes to the variables and files in a data structure to be sent to the Builder's eStream Packaging component. In the following paragraphs, detailed description of the Installation Monitor is described for Microsoft Windows environment.

In Microsoft Windows system, the Installation Monitor (IM) component consists of a kernel-mode driver subcomponent and a user-mode subcomponent. The kernel-mode driver is hooked into the Windows registry and file system function interface calls. The hook into the registry function calls allows the IM to monitor system variable changes. The hook into the file system function calls enables the IM to observe file changes.

The IM kernel-mode (IM-KM) driver subcomponent is controlled by the user-mode subcomponent (IM-UM). The IM-UM sends messages to the IM-KM to start and stop the monitoring process via standard I/O control messages called IOCTL. The IM-KM memorizes any addition or deletion of registry variables. It also records changes to

application-specific, shared among a group of applications, or system-wide files. Every files and directories are assigned a unique file number for simplifying identification of a specific file. Once the installation of an application completed, the IM-UM retrieves these changes from the IM-KM and forward the data structure to the eStream Packager.

Builder Install Monitor Control Flow Diagram



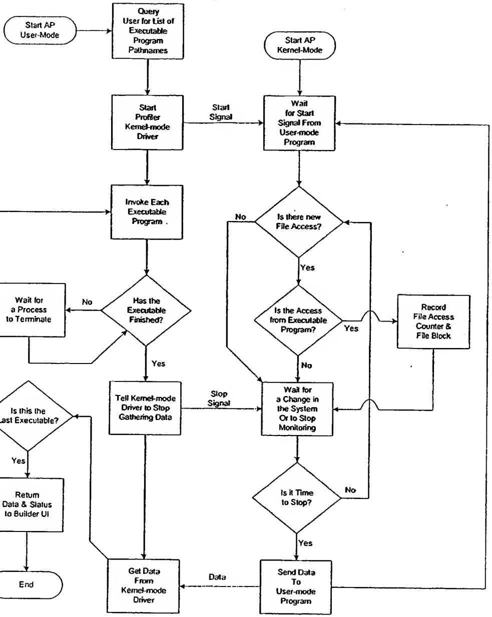
Application Profiling
In the second phase of the conversion process, the Builder's Application Profiler (AP) component invokes the application executable program that is installed during the first phase of the conversion process. The executable program files are accessed in a particular sequence. And the purpose of the AP is to captures this sequence data. This data is useful in several ways.

First of all, frequently used file blocks can be streamed to the eStream client before other less used file blocks. A frequently used file blocks is cached locally on the eStream client cache before the user starts using the streamed application for the first time. This has the effect of making the streamed application as responsive to the user as the locally installed application by hiding any long network latency and bandwidth problems.

Secondly, the frequently accessed files can be reordered in the directory to allow faster lookup. This optimization is useful for directories with large number of files. When the eStream client looks up a frequently used file in a directory, it finds this file early in the directory search. In an application run with many directory queries, the potential performance gain is significant.

The Application Profiler (AP) is not as tied to the system as the Installation Monitor (IM) but there is still some OS dependent issue. In the Windows system, the AP still has two subcomponents: kernel-mode (AP-KM) subcomponent and the user-mode (AP-UM) subcomponent. The AP-UM invokes the converting application executable. Then AP-UM starts the AP-KM to track the sequences of file block accesses by the application. Finally when the application exits after the desired amount of sequence data is gathered, the AP-UM retrieves the data from AP-KM and forwards the data to the eStream Packager.

Builder Profiler Control Flow Diagram



EStream Packaging

In the final phase of the conversion process, the Builder's eStream Packager (EP) component processes the data structure from IM and AP to create a data set suitable for streaming over the network. This converted data set is called the eStream Set and is suitable for uploading to the eStream Servers.

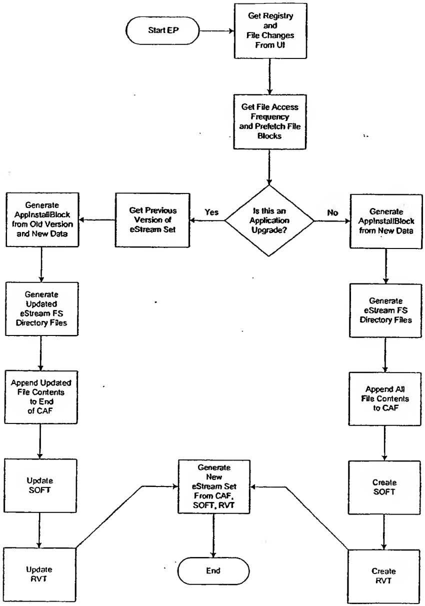
The eStream Set consists of the three sets of data from the eStream Server's perspective. The three types of data are Concatenation Application File (CAF), Size Offset File Table (SOFT), and Root Versioning Table (RVT).

The Concatenation Application File (CAF) consists of all the files and directories needed to stream to the client. The CAF can be further divided into two subsets: initialization data set and the runtime data set. The initialization data set is the first set of data to be streamed from the server to the client. This data set contains the information captured by IM and AP needed by the client to prepare the client machine for eStreaming this particular application. This initialization data set is also called the AppInstallBlock. Detailed format description of the AppInstallBlock is described in another document. The second part of the CAF consists of the runtime data set. This is the rest of the data that is streamed to the client once the client machine is initialized for this particular application. The EP appends every files recorded by IM into the CAF and generates all directories. Each directory contains list of file name, file number, and the metadata associated with the files in that particular directory.

The EP is also responsible for generating the SOFT file. This is a table used to index into the CAF for determining the start and the end of a file. The server uses this information to quickly access the proper file within the directory.

Finally, the EP creates the RVT file. The Root Versioning Table contains a list of root file number and version number. This information is used to track minor application patches and upgrades. The EP generates new directories when any single file is changed from the patch upgrade. The RVT is uploaded to the server and requested by the eStream client at appropriate time for the most updated version of the application by a simple comparison of the client's eStream application root file number with the RVT table located on the server.

Builder eStream Packager Control Flow Diagram



Data Flow Description

The following list describes the data that is passed from one component to another. The numbers corresponds to the numbering in the Data Flow diagram.

1. The full pathname of the installer program is query from the user of the Builder program and is sent to the Install Monitor.

2. The Install Monitor (IM) user-mode sends a read request to the hard-drive controller to spawn a new process for installing the application on the local machine.
3. The OS loads the application installer program into memory and run the installer program.
4. The installer program reads more files from the CD media.
5. The CD media data files are read into memory by the installer program.
6. The application installer program writes the files into proper locations on the local hard-drive.
7. IM kernel-mode captures all file read/write requests and all registry read/write requests by the installer program.
8. IM kernel-mode program sends the list of all file changes and all registry changes to the IM user-mode program.
9. IM user-mode identify special files which needs to be copied or spoofed into eStream client machine before the regular files can be streamed. It also assigns unique file numbers to every file. This data is returned to the Builder UI.
10. Builder UI invokes Application Profiling (AP) user-mode program by querying the user for the list of application executable names to be profiled.
11. Application Profiler user-mode invokes each application executable in succession by spawning each program in a new process.
12. The OS loads the application executable into memory and run.the executable.
13. The executable file image is loaded into memory and starts executing. The application files will continuously be loaded into memory as needed.
14. Every file accesses to load the application file blocks into memory is monitored by the Application Profiler (AP) kernel-mode.
15. Application Profiler kernel-mode returns the file access sequence and frequency information to the user-mode program.
16. Application Profiler returns the processed profile information. This has two sections. The first section is used to identify frequency of files accessed. The second section is used to list the file blocks for prefetch to the client.
17. The eStream Packager receives files and registry changes from the Builder UI. It also receives the file access frequency and a list of file blocks from the Profiler.
18. The eStream Packager reads all file data from the hard-drive that are copied there by the application installer.
19. The eStream Packager also reads the previous version of eStream Set for support of minor patch upgrades.
20. Finally, the new eStream Set data is stored back to non-volatile storage.

Mapping of Data Flow to eStream Set

Step 7: Data gathered from this step consist of the registry and file changes. This data is mapped to the AppInstallBlock's File Section, Add Registry Section, and Remove Registry Section.

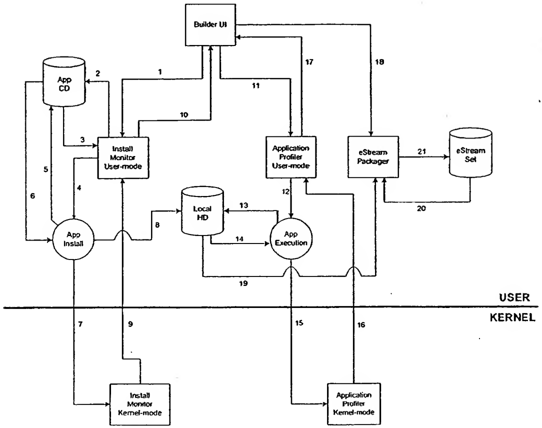
Step 8 & 19: File data are copied to the local hard-drive then concatenated into part of the CAF contents.

Step 10: Data returned to the Builder UI contains unique file numbers. This data is mapped to the file numbers used throughout the eStream Set data structure.

Step 15: Part of the data gathered by the Profiler is used to generate a more efficient eStream FS Directory content. Another part of the data is used in the AppInstallBlock as a prefetch hint to the eStream client.

Step 20: If the installation program was an upgrade, eStream Packager needs previous version of the eStream Set data. Appropriate data from the previous version is combined with the new data to form the new eStream Set.

eStream Builder Data Flow Diagram



Format of eStream Set

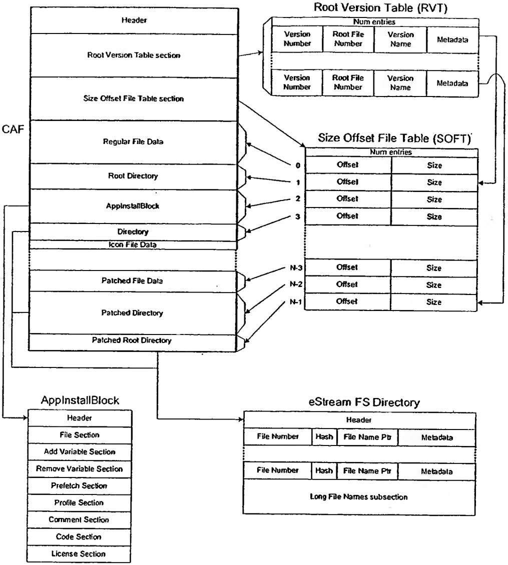
The format of the eStream Set consists of 3 sections: Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF). The RVT section lists all versions of the root file numbers available in an eStream Set. The SOFT section consists of the pointers into the CAF section for every file in the CAF. The CAF section contains the concatenation of all the files. The CAF section is made up of regular application files, eStream FS directory files, AppInstallBlock, and icon files. Please see the document on eStream Set Format for detailed format of the eStream Set.

OS dependent format

The format of the eStream Set is designed to be as portable as possible across all OS platforms. At the highest level, the format of CAF, SOFT, and RVT that make up the format of eStream Set are completely portable across any OS platforms. The only critical piece of data structure that is OS dependent is located in the initialization data set called AppInstallBlock in the CAF. This data is dependent on the type of OS due to the differences in low-level system differences among different OS. For example, the Microsoft Windows contain system environment variables called the Registry. The Registry has a particular tree format not found in other operating systems like UNIX or MacOS.

Another OS dependent format is the format of the file names. Applications running on the Windows environment inherit the old MSDOS 8.3 file name format. To support this properly, the format of the Directory file in CAF requires an additional 8.3 field. This field is not needed in other operating systems like UNIX or MacOS.

Format of the eStream Set



v 0.1

Device driver versus file system paradigm

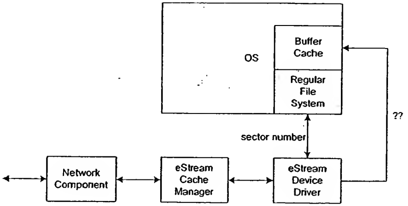
The eStream Prototype is implemented using the 'device driver' paradigm. One of the advantages of the device driver approach is that the caching of the sector blocks is

simpler. The client cache manager only needs to track sector number in its cache. In comparison with the 'file system' paradigm, more complex data structure is required to track a subset of a file that is cached on a client machine. This makes 'device driver' paradigm easier to implement.

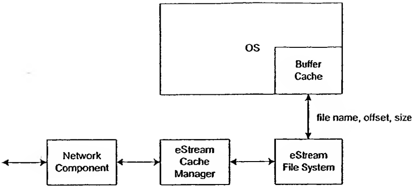
On the other hand, there are many drawbacks to the 'device driver' paradigm. On the Windows system, the device driver approach has problem supporting large number of applications. This is due to the limitation on the number of assignable drive letters available in a Windows system (26 letters); and the fact that each application needs to be located in its own device. Note that having multiple applications in a device is possible, but then the server needs to maintain exponential number of devices that support all possible combinations of applications. This is too costly to maintain on the server.

Another problem with the device driver approach is that the device driver operates at the disk sector level. This is a much lower level than operating at the file level in the file system approach. The device driver does not know anything about files. Thus, the device driver cannot easily interact with the file level issues. For example, spoofing files and interacting with OS buffer cache is nearly impossible with device driver approach. But both spoofing files and interacting with OS buffer cache is need to get higher performance.

Device Driver Paradigm



File System Paradigm



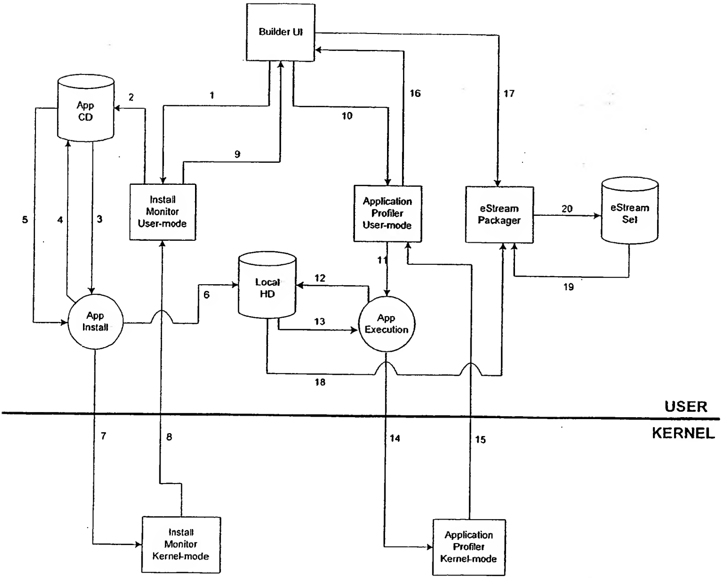
Implementation in the Prototype
The prototype has been implemented and tested successfully on the Windows and Linux distributed system. The prototype is implemented using the 'device driver' paradigm as described above. The exact procedure for streaming application data is described next.

First of all, the prototype server is started on either the Windows or Linux system. The server creates a large local file mimicking large local disk images. Once the disk images are prepared, it listens to TCP/IP ports for any disk sector read or write requests.

Secondly, the conversion process is done on a Windows system via semi-manual procedure. The server disk image is 'mounted' on the local Z drive by making the proper TCP/IP connection to the server. Then the application installation program is invoked and the application is installed into the Z drive. This writes the application files into the Z drive device driver, through the TCP/IP connection, and finally on to the server disk image. At the same time, a file and registry monitoring program records all registry and file changes. This data is stored as an initialization file to be invoked on the client to prepare the client machine for streaming.

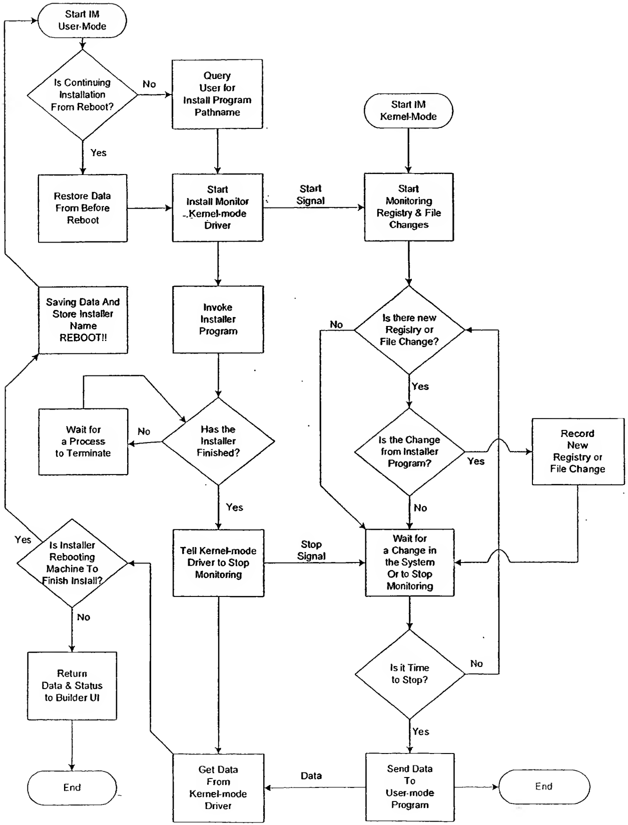
Finally, after the application files is stored on the server disk image, the client prototype is started. The client connects to the server and 'mount' the server disk image as a local Z drive. Then the initialization file is invoked which setup the local registry variables and copy system files into proper directories. Once the local machine is prepared for streaming that particular application, the user can start using the application. When the application is first started, the pages are not located in the local buffer cache. The OS makes sector request to the eStream device driver that forwards the sector request to the eStream Cache Manager. If the sector is located in the eStream cache, then the data is returned immediately. If the data is not located in the eStream cache, then the request forwarded to the network component that sends the message to the server. The server finds the proper sector data and returns the data to the client. The client eStream Cache Manager caches the new sector data and forwards the sector data to the eStream device driver. The device driver returns the sector data to the OS.

eStream Builder Data Flow Diagram

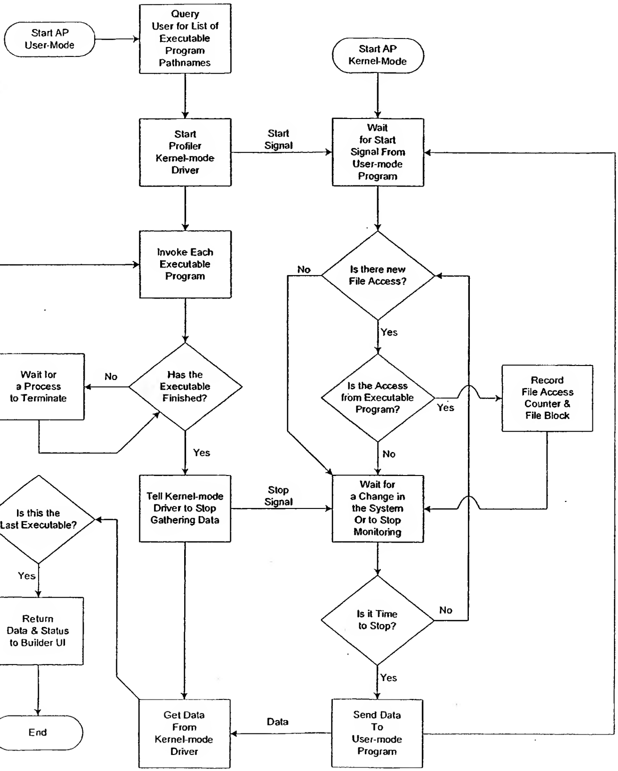


v 0.1

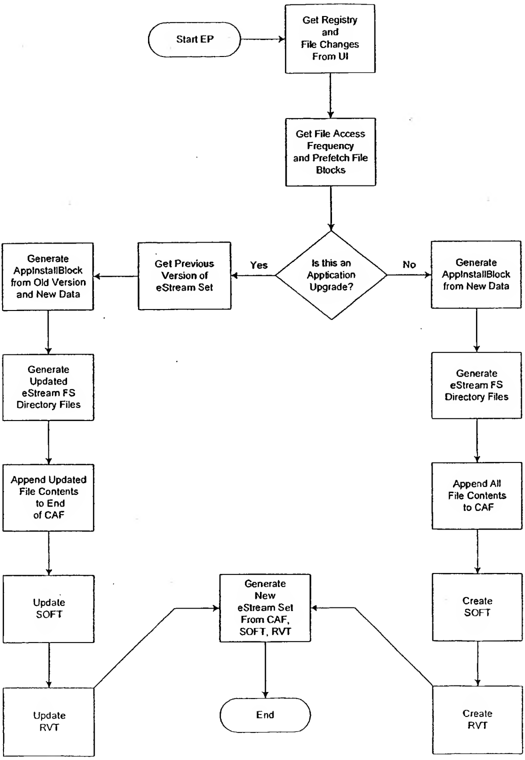
Builder Install Monitor Control Flow Diagram



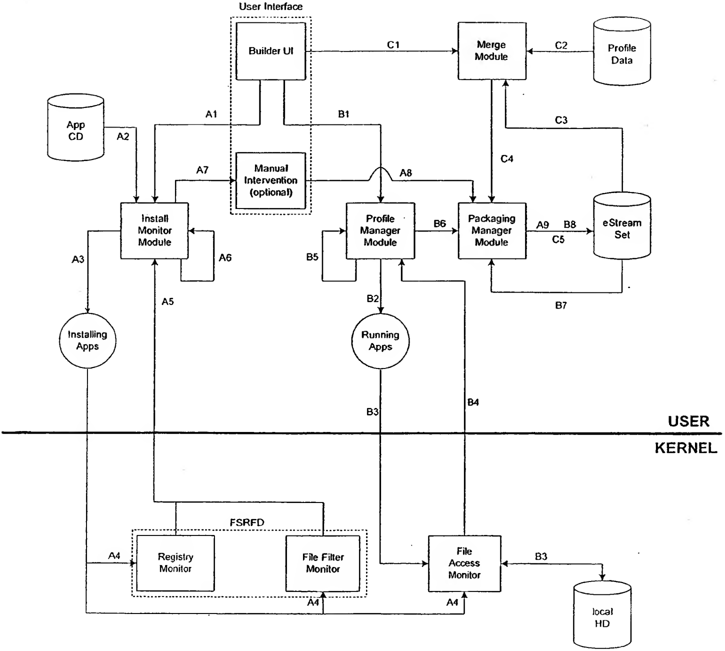
Builder Profiler Control Flow Diagram



Builder eStream Packager Control Flow Diagram



eStream Application Builder High-Level Design Diagram



v 0.1

eStream Builder File Access Monitor Low Level Design

Sanjay Pujare and David Lin
Version 0.1
[redacted]

Functionality

The eStream Application Builder File Access Monitor (FAM) is a kernel-mode device driver that behaves as a file filter driver to has the following responsibilities:

- Monitor any running application's request to access a file or directory
- Track application file and directory accesses
- Track file metadata queries
- Start and stop profiling via IOCTL requests from the user-mode program
- Return the file access data to the user-mode program via I/O Request Packet (IRP)
- Return any error conditions to the user-mode program via IRP

The File Access Monitor is based on the 'Filemon' program. The source code for the program is available free for download over the Web at http://www.sysinternals.com/filemon.htm.

Data type definitions

The File Access Monitor (FAM) monitors a sequence of file block accesses by a particular process or one of its child processes. The FAM also tracks any queries on the file metadata. The combination of the file content and metadata is returned to the Profile Manager for further processing. The following is the data structure externally visible to the other subcomponents outside FAM.

```
Struct SequenceData
{
   UINT NumEntries;
   Struct Entry
   {
      PUNICODE_STRING FilePathName;
      BOOL IsAccessingMetadata;
      ULONG Offset;
      ULONG Size;
   } Entries[NumEntries];
};
```

The FileName contains the null terminating string of the file accessed. IsAccessMetadata flag indicates if the access is on the file metadata or the file content. If the operation is on

the file content, then the fields 'Offset' and 'Size' indicate the location of the read or write operations. Otherwise, the fields 'Offset' and 'Size' are not used.

Interface definitions

Function 1 : Hooks into user defined IOCTL calls

```
// The following is a Fast I/O Device Control
// call interface.  Each of the user IOCTL
// call to the driver is described here.
// The OICTL input and output parameters are
// stored on the IRP on the InputBuffer and
// OutputBuffer respectively.
// This function must be called only by NT I/O
// Manager.

BOOLEAN FileSysFastIoDeviceControl(
   IN PFILE_OBJECT FileObject,
   IN BOOLEAN Wait,
   IN PVOID InputBuffer,
   IN ULONG InputBufferLength,
   OUT PVOID OutputBuffer,
   IN ULONG OutputBufferLength,
   IN ULONG IoControlCode,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Input:
   IoControlCode==IOCTL_FAM_VERSION
      OutputBuffer: version number of the driver

   IoControlCode==IOCTL_FAM_START
      InputBuffer: process ID to monitor

   IoControlCode==IOCTL_FAM_STOP
      OutputBuffer: stop profiling

   IoControlCode==IOCTL_FAM_GETDATA
      OutputBuffer: get sequence data

   IoControlCode==IOCTL_FAM_GETSTATUS
      OutputBuffer: get status from driver

Output:
```

```
Comments:
   The IOCTL calls from the user program to
   the device driver is either through the
   dispatcher or through the Fast I/O
   interface.

Errors:
```

Function 2: DriverEntry

```
// Called by the NT system to initialize
// driver.  The following entries are hooks
// into the OS and are not called by any of our
// component directly.
NTSTATUS DriverEntry(
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE_STRING RegistryPath)

Comments:
   Initialize the driver
```

Function 3: Hooks into Fast I/O functions

```
// NT Fast I/O calls.  These are some of the
// hooks into the OS
NTSTATUS FileSysFastIoRead(
   IN PDRIVER_OBJECT DriverObject,
   IN PLARGE_INTEGER FileOffset,
   IN ULONG Length,
   IN BOOLEAN Wait,
   IN ULONG LockKey,
   OUT PVOID Buffer,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Comments:
   Hooks into Fast I/O Read
```

Function 4: Hooks into dispatcher functions

```
// Besides hooks into Fast I/O calls, we
// must also hook into each of the major
// functions like IRP_MJ_CREATE, IRP_MJ_READ,
// etc...
FileSysHookRoutine(
   PDEVICE_OBJECT HookDevice,
```

```
IN IRP Irp)
```

Component design

I/O Hook location

The trickiest part of the File Access Monitor (FAM) component design is determining the locations in the Operating System to hook the routines. FAM must provide driver entry function for initializing the driver. It must also provide hooks into the OS to monitor read and write file operations. In addition, it needs to monitor access to file metadata. And finally, it has to provide the user-mode program a way to communicate to the FAM through the IOCTL calls.

The FAM must behave like any other Windows kernel-mode drivers by exporting the standard *DriverEntry* function. The *DriverEntry* function has the following purposes:

- Check for OS build version.
- Setup the device name
- Call OS routine to create the device
- Make symbolic link to allow device access from Win32 programs
- Create dispatch points for all routines that must be handled
- Setup Fast I/O hooks
- Initialize all data structures

In addition to the *DriverEntry*, the FAM must handle five user defined IOCTL calls.

- IOCTL_FAM_VERSION
- IOCTL_FAM_START
- IOCTL_FAM_STOP
- IOCTL_FAM_GETDATA
- IOCTL_FAM_GETSTATUS

In IOCTL_FAM_START, the handler receives the process ID from the user-mode program. It uses this process ID to filter out relevant file and metadata accesses. In IOCTL_FAM_STOP, the handler stops monitoring and recording any file accesses. In IOCTL_FAM_GETDATA, the handler packages the file access sequence in the I/O Request Packet (IRP) to be returned to the user-mode program. Finally, in IOCTL_FAM_GETSTATUS, the handler returns its current status. This status includes: FAM_STATUS_OK, FAM_STATUS_ERROR, and FAM_STATUS_PROFILING.

In addition to the user defined IOCTL hooks, the FAM must add hook into both the dispatch points and the Fast I/O calls to monitor all read and write requests. In addition, the FAM monitors any metadata accesses. The following is a list of Fast I/O calls it must hook:

- FastIoRead
- FastIoWrite

- o FastIoMdlReadComplete
- o FastIoMdlWriteComplete
- o FastIoReadCompressed
- o FastIoWriteCompressed
- o FastIoQueryBasicInformation
- o FastIoQueryStandardInformation

In the routine to handle FastIoRead and FastIoWrite, the driver must determine the process ID making this request. If the process is in the list of monitoring processes, the file name, file offset, and size is recorded and added to the profile sequence list. In the routine to handle FastIoQueryBasicInformation and FastIoQueryStandardInformation, the driver records the file name associated with this metadata query.

In addition to hooks to the Fast I/O calls, the I/O may call the File System services through standard Windows NT dispatch points. The following is a list of dispatch points to be handled by FAM:

- o IRP_MJ_CREATE
- o IRP_MJ_READ
- o IRP_MJ_WRITE
- o IRP_MJ_DIRECTORY_CONTROL + IRP_MN_QUERY_DIRECTORY
- o IRP_MJ_QUERY_INFORMATION
- o IRP_MJ_SET_INFORMATION
- o IRP_MJ_QUERY_EA
- o IRP_MJ_SET_EA

The routine to handle IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MN_QUERY_DIRECTORY is handled by the same function as the routine for handling FastIoRead and FastIoWrite. The routine to handle IRP_MJ_QUERY_INFORMATION, IRP_MJ_SET_INFORMATION, IRP_MJ_QUERY_EA, and IRP_MJ_SET_EA are handled by the same function as the routine for handling FastIoQueryInformation.

Communication with user-mode component (Profile Manager)

Besides using the IOCTL to send profile data to the Profile Manager, the FAM must also signal the Profile Manager when new data is available for retrieval. The Profile Manager wakes up from the signal by the FAM and retrieves the information on the blocks of files accessed. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWaitForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Process Filtering

The FAM must filter the profile information so only relevant data relating to the application under profiled is obtained. This is accomplished by filtering the data according to

the process ID invoking the file access operations. When the FAM is started, the Profile Manager sends its process ID. FAM assumes all child processes of the Profile Manager process ID is to be monitored since the Profile Manager invokes all applications using *CreateProcess()* API. Thus, the new processes all inherit Profile Manager as its parent process ID. The process filtering is accomplished using *PsSetCreateProcessNotify-Routine()* to add a hook to the OS. FAM is notified whenever there is a new process created. The process ID is recorded in a list if its ancestor is the Profile Manager process ID. This list is used to filter the profile data gathered by FAM.

Locks

Since multiple threads may be entering different sections of FAM and accessing different data structures, appropriate locks must be used to prevent multiple threads from reading and writing at the same time. *ExInitalizeResourceLite(), ExAcquireResourceExclusiveLite(),* and *ExReleaseResourceLite()* are used when shared data structure is accessed. These APIs have the requirement that the kernel APCs must be disabled before calling and that IRQL must be lower than DISPATCH_LEVEL. This can be accomplished by using *KeEnterCriticalRegion()* and *KeLeaveCriticalRegion()*. The following is a sample code using these APIs:

```
ERESOURCE gResource; // global variable

KeEnterCriticalRegion()
ExAcquireResourceExclusiveLite(&gResource, TRUE);

<critical section of code>

ExReleaseResourceLite(&gResource);
KeLeaveCriticalRegion();
```

Testing design

- **Unit testing plans**

The plan for unit testing of the FAM consists of using the Profile Manager (PM) and a File Access Driver (FAD) as the test drivers. The PM tests user-defined IOCTL calls. The FAD creates desired data pattern from the OS's I/O Manager to the FAM. The FAD tests the FAM's ability to monitor file accesses by querying files and directories in a particular order. Together, the PM and FAD test coverage of the FAM is complete. The following is a list of tests:

1. Test each user-defined IOCTL interface via PM by sending border cases.
2. Test to make sure FAM captures every file and directory access via standard file I/O requests from a user-mode program called FAD.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

Other Builder components log error messages to a predefined file. The kernel-mode programs do not have the capability to read/write to a file. Since FAM is a kernel-mode program, an alternative method of reporting error messages has to be developed. Current, the FAM has a user-defined IOCTL interface (IOCTL_FAM_GETSTATUS) to retrieve the error messages. FAM keeps a stack of error messages encountered and reports the stack of error messages at the request by an appropriate user-mode program.

Open Issues

- Exactly which Fast I/O calls need to be hooked to get all the read and write operations for file accesses?
- Along the same line, which dispatch points need to handled to get all the read and write operations for file accesses?
- Have we hooked into all possible places where the metadata accesses can occur?
- Does the FAM need to hook into FileLock and FileUnlock operations?

eStream Application Builder High-Level Design

Authors: Sanjay Pujare and David Lin

Version 0.1

[redacted]

This document contains the high level design of the eStream Application Builder. The Builder is used to "prepare" an application before it can be eStreamed. This document describes the high level design of the application installation monitoring, file relocation and mapping, gathering of the initial profiling information of an application, the packaging of the eStream Set, and the merging of the newly uploaded user profile data.

Note: all references to "user" should be understood to mean the user of the Builder (i.e. the person who is responsible for creating eStream sets) and not the end-user of eStream technology.

This document described these steps involved in the preparation of the application: Installation Monitoring, Application Profiling, and eStream Packaging.

Modules

Installation Monitor:

- When the application is installed, we need to monitor the installation to see various "things" taking place on the computer. These could be:
  - Various updates to the System Registry
  - Files added to the Install directories (i.e. directories where application bits are copied as specified by the installing user). Lets call this group F_I.
  - Files added/updated to the Shared directories (e.g. "Program Files\Common Files"). Lets call this group F_C.
  - Files added/updated to the System directories (e.g. "WinNT\System32"). Lets call this group F_S.
  - Files added/updated to the User specific directories (e.g. "Documents and Settings\spujare\Application Data"). Lets call this group F_U.

  Note that once this information is gathered by the "Installation Monitor", a single "Installation Set" is prepared where all the files are stored in a single directory hierarchy. Note that files in the F_C, F_S and F_U groups (i.e. F_{CSU} group) are also stored here. For these files a "mapped location" is created under the single directory hierarchy. The Installation Set typically creates a map of all files (called ISM for Installation Set Map) described above with each entry containing the following info:

  1. fileId for the file
  2. location and name of the file. Note that the location will be the actual location for F_I files, but mapped location for the F_{CSU} files.

- After we gather the above information, we need to prepare a "File Relocation Map" (FRM) that is used by the client file spoofer to spoof references to any file in the common file group (i.e. F_{CSU}). For example: when the eStreamed app makes a reference to a file `C:\Program Files\Word\Foobar`, the file spoofer actually redirects that reference to `Z:\Program Files\Word\Foobar`. It does that because of the File Relocation Map. Each entry in the FRM typically has the following info:
  1. fileId (which references an entry in the ISM).
  2. Actual location where the application expects it (i.e. `C:\Program Files\Word\Foobar`).

Profile Module:
During the application building process, the Builder program queries the user for the name of the application executable. Then Builder program starts and terminates the application executable immediately to gather initial sequence of the application page access pattern. After the initial seed of profile data is acquired, the Profile Sequence Matrix is combined with other appInstallBlock data gathered from the Install Monitor.

Profile Sequence Matrix is a 2D matrix of a profile data. Each entry of the matrix [column C, row R] is an integer value indicating the number of times a page R is requested following the request of page C. This successor request pattern is the page requests missed in the eStream cache manager.

Package Module:
In the final phase of the Builder program, the appInstallBlock is encapsulated into a special installation executable and the application files is archived into a single compressed package. The install executable containing the appInstallBlock and the archive of application files can then be placed in a suitable eStream Set server for ASP to download to their machines.

Merge Module: (not supported in version 1.0)
During normal eStream application usage, the eStream client gathers profile information for that particular run of the application. Then at the termination of an eStream application, it uploads the new Profile Sequence Matrix to the Profile Server. The clients should not upload the Profile Sequence Matrix from previous runs because the Profile Server has no mechanism for distinguishing between previously uploaded data and the newly acquired data.

At appropriate time, the Builder is invoked to merge the newly uploaded per-user Profile Sequence Matrix into a collective Matrix. The merging algorithm may be designed with some heuristics to prevent the data biasing toward power users. This collective Matrix can be reinserted into the appropriate appInstallBlock then downloaded by any requesting eStream clients.

Kernel Device Drivers: